*Signal Processing in Databases*

# RetractorDB

## *User Manual*

*@Muro/MichalWidera*

# Contents

# Chapter 1

# RetractorDB

This chapter is a map, not a catalog. Instead of listing everything ever written about streams and signals, I show five strands of peer-reviewed literature at whose intersection RetractorDB sits, and for each of them I answer three questions: what has this strand already solved, how does RetractorDB differ from it, and what does this strand **not** touch. Only by overlaying these five layers does the gap this project fills become visible.

**Info**

retractordb.pdf — generated automatically on every `git push`.

**Note**

This system is an: Edge Signal Processing Engine. RetractorDB supports — rather than replaces — time-series databases (TSDB) and data stream management systems (DSMS): it works close to the signal source, pre-processes and filters high-frequency measurements using a declarative query language, keeps a partial, correctable record of past events and scheduled future ones in inspectable artifacts, and passes exact, deterministic results up the architecture — so that only reduced, already-processed streams reach the central architecture.

**Info**

Why did I place this chapter so early? Because an honest answer to the question "is this needed?" first requires showing what already exists. Most ideas in computer science have already been thought of once — reinventing the wheel wastes someone else's effort. This chapter is my attempt to prove that this particular wheel has not, in fact, been invented yet.

## 1.1. Five neighboring fields

The problem RetractorDB solves does not belong entirely to any single discipline. It sits in the gap between five:

1. **Number theory** – Beatty sequences, Fraenkel's theorem, covering systems. This provides the formal foundation.
2. **Task scheduling via Beatty sequences** – the same mathematics, a different application. The closest application-level neighbor.
3. **Digital signal processing (DSP)** – nonuniform sampling and filter banks with rational coefficients. This is the DSP counterpart of the interleaving operation.
4. **Data stream management systems (DSMS)** – stream algebras and continuous-query semantics. This is the database reference point.
5. **Time-series systems (TSMS) and in-database DSP** – the narrowest, most sparsely populated niche, closest to the system's actual goal.

I discuss them in turn, from the foundation toward the application.

## 1.2. Number theory: Beatty sequences and covering systems (1)

RetractorDB's entire algebra rests on the Beatty sequence and its generalization by Fraenkel to rational numbers. I cite these results in Formal Foundations and Proofs. Here I'm interested in the broader backdrop: how this mathematics functions in the contemporary literature, and whether anyone has already applied it where I have.

Beatty sequences have a rich combinatorial literature and documented applications in aperiodic tilings (quasicrystals), periodic scheduling, computer vision (digital lines), and formal language theory [11]. The strand is alive: Schaeffer, Shallit, and Zorcic (2024) showed that a non-homogeneous Beatty sequence is synchronizable by a finite automaton, which leads to decidability of the first-order theory of these sequences [12]. For me, however, the most relevant work is that of Berger, Felzenbaum, and Fraenkel (1986) on disjoint covering systems based on **rational** Beatty sequences [13] — exactly the variant my de-interleaving is based on, and one I did not cite in the original paper.

**What this strand does not touch:** number theory studies these sequences as mathematical objects. It does not connect them to a database, to a stream-processing model, or to signal processing. It supplies bricks, not a building.

## 1.3. Task scheduling via Beatty sequences (2)

This is the strand I must discuss most honestly, because it uses **the same proof machinery** as my theorems — just for a different purpose. In the periodic-scheduling problem (so-called *pinwheel scheduling*), tasks with different repeat periods are distributed so that tasks with one repeat time land in time slots belonging to the first complementary Beatty sequence, and those with the other, into the second [14]. Recent work (2025) proves results on the Rayleigh/Beatty partition using identities on floor and ceiling functions of the type $\lceil (m+l)a \rceil - \lceil ma \rceil$ [15] — almost identical, point for point, to the apparatus in my proof that de-interleaving satisfies Fraenkel's postulates.

The conclusion, for me, is twofold. On one hand — this is independent confirmation

that the approach is correct and natural; if someone else arrives, by the same road, at a working scheduling scheme, the foundation is solid. On the other — it narrows what I can call novel. “Beatty sequences for scheduling” already exists and is being actively published. Interestingly, my system uses this mathematics **internally**, precisely for task scheduling (see Query Execution) — but that’s not where the original contribution lies.

**What this strand does not touch:** scheduling treats sequences as a tool for allocating time slots to processors. It doesn’t build a data algebra on top of them, doesn’t use them to express operations on signals, and doesn’t create a query language.

## 1.4. Digital signal processing: nonuniform sampling and filter banks (3)

The interleaving and de-interleaving operation is, in DSP terms, a sample-rate conversion between streams with different Δ. Here a broad, mature literature exists. The closest bridge is the work of Samadi, Ahmad, and Swamy (2004), which formulates the perfect-reconstruction condition for nonuniform filter banks based on the system’s response to delayed unit-step signals [16] — thereby introducing step-function (and, indirectly, floor-function) machinery into the domain of multirate DSP. The broader strand is periodic-nonuniform sampling of band-limited signals [17], and — directly relevant — filter banks with **rational** decimation factors (Kovačević and Vetterli) [18].

Number-theoretic constructions even show up there: Ramanujan filter banks extract periodic components of a signal [19]. But I have not found Beatty sequences or Fraenkel’s theorem specifically in this literature — and that’s part of the gap.

**What this strand does not touch:** DSP operates in the z-domain, the frequency domain, on frames and bases. It doesn’t treat resampling as a declarative algebraic operator, nor does it embed it in a database system. Coefficients are sometimes rational, but the apparatus is analysis, not the number theory of set partitioning.

## 1.5. Data stream management systems (DSMS) (4)

On the database side, the canon is CQL from Stanford’s STREAM project (Arasu, Babu, Widom). In this model, a stream is a potentially infinite multiset of elements $\langle s, \tau \rangle$, where s is a tuple and τ a timestamp [20]; query semantics is built on windows and stream↔relation mappings. A second close neighbor is the temporal algebra of Krämer and Seeger (the PIPES system), providing deterministic results for continuous queries and a rich set of transformation rules underlying optimization [21].

This is the proper reference point for my algebra and my expression rewrite rules. The difference, however, is fundamental and concerns the data model itself. CQL and PIPES build their semantics on the (s, τ) model — every tuple carries its own timestamp, and operators act through windows. I adopt a differential model ($s_n$, Δ), with a rational, fixed value of Δ per stream, and I derive the operators that align streams with different Δ from number theory. This is not a cosmetic difference in syntax —

it's a different data model, leading to a different class of operators (interleaving, de-interleaving) and a different optimization method.

In deployment terms, the relationship is complementary rather than competitive: RetractorDB acts as an edge-level pre-processing and buffering stage, whose exact, deterministic results can feed a windowed DSMS.

**What this strand does not touch:** DSMS aim at approximate, scalable processing of unbounded streams, tolerant of out-of-order timestamps. They don't aim for exact, deterministic DSP operations under strict time discipline, and they don't reach for number theory for resampling semantics.

## 1.6. Time-series systems (TSMS) and in-database DSP (5)

This is the narrowest niche — and the closest to RetractorDB's actual goal. The canonical survey is Jensen, Pedersen, and Thomsen's "Time Series Management Systems: A Survey" (IEEE TKDE, 2017) [22]. The Plato system described there is the closest real "DSP inside a database": it combines an RDBMS with signal-processing methods, eliminating the need to export data to external tools like R or SPSS [22]. The other approaches to "signals in a database" boil down to approximation and compression — wavelet, dictionary, and shape-based representations.

All of them, however, treat DSP as approximation or after-the-fact analytics. None makes signal-processing operations **exact, deterministic first-class operators** within a query algebra. This confirms that the niche is thin, and that my angle of attack — exactness over rational numbers — is distinct.

**What this strand does not touch:** TSMS optimize ingestion scale, compression, and retention. DSP is a second-class citizen in them — an analytical add-on, not the core of the semantics.

## 1.7. The blank spot: where the contribution lies

Once the five layers are overlaid, the picture becomes clear. Each field touches one or two walls of the problem, but **none occupies their intersection**:

| Field | Beatty/Fraenkel | Exact DSP | Stream algebra / query language | Deterministic time discipline |
|---|---|---|---|---|
| Number theory | ✓ | – | – | – |
| Scheduling (pinwheel) | ✓ | – | – | partial |
| Multirate DSP | – | ✓ | – | – |
| DSMS (CQL, PIPES) | – | – | ✓ | – |
| TSMS / in-database DSP | – | partial | partial | – |
| **RetractorDB** | ✓ | ✓ | ✓ | ✓ |

RetractorDB's contribution lies not in any single component — it lies in their **synthesis**: in using covering systems (rational Beatty sequences and Fraenkel's theorem) as the semantic foundation for a declarative stream algebra that implements exact signal-processing operators inside a database system, under deterministic time discipline. A clarification matters here: the system guarantees deterministic execution **semantics** (identical inputs give identical results in identical order) and a predictable, sequential execution model — I deliberately do not claim hard real-time guarantees, since those require worst-case execution-time analysis on a real-time operating system, and remain future work. Number theory has Beatty, and even scheduling, but doesn't connect them to a database or to DSP. DSP has multirate processing and rational filter banks, but doesn't reach for Fraenkel and doesn't frame it as a query language. DSMS has stream algebras and optimization rules, but on the windowed $(s, \tau)$ model, not the differential $(s_n, \Delta)$ one. This intersection is empty.

**Warning**

Hence a real risk, which I point out directly: the scheduling community has been publishing this same Beatty/Fraenkel machinery in 2023–2025. The problem itself — together with the need for a declarative stream algebra and a continuous query language — was already formulated back in 2003–2005, in the context of computer-assisted fetal monitoring [25]; I laid the "covering systems ↔ stream alignment and DSP" bridge in a 2006 publication [3], but in a venue with low discoverability. If this result doesn't reach well-cited circulation, the same bridge may be independently built and credited to someone else.

## 1.8. Methodological caveat

This is a targeted review, not a systematic one — based on searching across five strands, not a full citation analysis. A "forward citation" review of Samadi's paper [16] confirms the point: according to Semantic Scholar (as of July 2026), its only recorded citations are a paper on Gabor window design, two systems-theoretic papers on multirate systems, and the 2006 bridge paper itself [3] — none of them uses Beatty sequences or Fraenkel's theorem. The closest use of this machinery outside number theory that I'm aware of is the construction of exponential Riesz bases from Beatty–Fraenkel sequences (Pfander, Revay, and Walnut) [24] — but that belongs to pure harmonic analysis and doesn't touch filter banks or sample-rate conversion. What remains for full closure is a systematic review of the scheduling strand [14] and of the filter-bank literature as a whole; if a use of Fraenkel's theorem in multirate DSP exists, it narrows the scope of the novelty claim and should be accounted for here.

# Chapter 2

# Mathematical Foundations

## 2.1. Mathematical Foundations

**Info**

Do you know what the Fields Medal is? It is an award given exclusively to outstanding mathematicians under the age of 40. It is called the mathematical Nobel Prize. Interestingly, no mathematician will ever receive the actual Nobel Prize — as per the founder's wishes. John Charles Fields himself (1863-1932) was a Canadian mathematician. John Charles Fields had one doctoral student — Samuel Beatty (1881-1970).

In 1926, Samuel Beatty published the following theorem [1]:

If p, q are positive irrational numbers satisfying the relation

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

then the sequences

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

and

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

partition the set of positive integers.

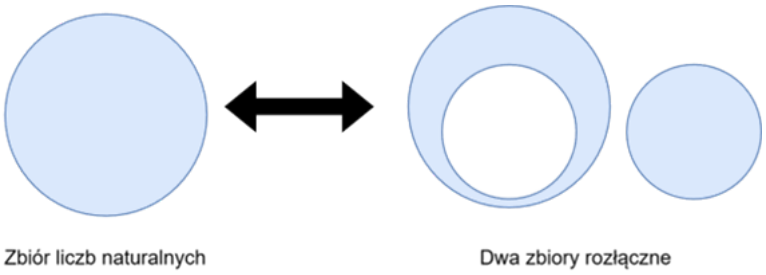



Fig. 1. Graphical representation of the concept of disjoint sets

These two sequences partition the set of natural numbers. This means that given two irrational numbers satisfying the relation stated in the theorem, we can split the entire set of natural numbers into two disjoint sets (Fig. 1).

The Beatty theorem is a fascinating observation in its own right — but in computer systems we run into a problem with irrational numbers. Real numbers — despite the fact that some programming languages use the words Real or Float for the real-number type — have little in common with actual real numbers. The fundamental problem is that we don't have them, and presumably never will.

And here our journey would have abruptly ended, were it not for another theorem. The situation changed dramatically thanks to a mathematician — Aviezri Siegmund Fraenkel (1926), who specializes in combinatorial aspects of game theory.

In 1969 he presented the following theorem [2]. The starting point is a parameterized Beatty sequence:

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

This single definition generates an entire family of sequences. The theorem always concerns a **pair** of its instances with different parameters: the sequence

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

These sequences partition the set $\mathbb{N}$ if and only if the following five conditions are satisfied:

1.

$$0 < \alpha < 1$$

2.

$$\alpha + \beta = 1$$

3.

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

4. If α is an irrational number, then:

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

and

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

for

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

5. If α is a rational number (let q∈N be the smallest number such that qα∈N), then

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

and

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

And that is exactly what we need! We don't have irrational numbers, but rational numbers understood as the ratio of two natural numbers are something a computer can handle just fine.

In our case, I first built prototype equations in Python, and then started looking for mathematical foundations that looked similar and could serve as well-documented equations backed by formal proofs. Proofs, of course, carried out by more experienced mathematicians. Modest as my skills were, they were enough to identify these two publications as relevant to my ideas.

This document does not include formal proofs. That is why I present here only the equations and theorems actually used in the system. For the formal proofs, I refer the reader to my scientific publications [3].

## 2.2. Algebra of Regular Time Series

Algebra — understood as a construct consisting of a defined set and defined operations on it — forms the basis for the declarative query language developed here. Throughout the rest of this work, when referring to "the Algebra" (without a further qualifier) I mean the Algebra of regular time series. When I want to refer to Relational Algebra, I will state the qualifier explicitly.

I proposed [3] the following definition of a regular time series (the so-called data model), along with the following operations and definitions.

**Note**

By a data stream we understand an ordered pair S := (sn,Δ) — where the first element is an ordered series of data and the second, denoted by the symbol delta, is the regular time interval between consecutive elements of the series.

We adopt a fixed indexing convention: stream indices run from zero, and element sn carries an implicit, default timestamp of (n+1)·Δ. In other words — the first element of the stream appears after a full interval Δ has elapsed since the stream's creation. The timestamp is not carried within the tuple; it is derived from the position n and the rate Δ. It is precisely this differential data model that distinguishes the system from classical DSMS, where a stream is a multiset of pairs ⟨s,τ⟩ with a timestamp attached to every tuple.

A data series defined this way is referred to in the system as a data stream. Such a regularly flowing set of data passing through the system, usually described by a data schema, contains fields of various types. Each reading occurs at an equal time interval between consecutive measurements. This construction resembles a digital signal more than an irregular data stream — however, referring to it as a "stream" throughout the rest of the research will prove justified.

**Info**

Note:
The terms "stream" and "time series" are used interchangeably in this work and mean the same thing.
Formally, in the scientific literature a stream is denoted as a set of pairs (a,t) — where a denotes a tuple, and t denotes its moment of registration or occurrence. A stream allows tuples whose time t coincides for different tuples. In the case of a time series, we distinguish two types of series — regular and irregular.
- For irregular series — a series is a sequence of tuples ordered in time — {at,tn}, where the time tn is unique in the set for each tuple.
- A regular time series, on the other hand, can be described by a sequence of tuples and a regular time interval between their occurrences — ({at},D) — and it is this latter definition that forms the basis for further operations in the system developed here.

The operations that can be performed on such a data set are defined as follows:

- interleaving and de-interleaving
- sum and difference
- sequence shift
- aggregation and serialization

The interleaving operation involves two different data streams.

We define it as follows:

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

The arguments of the interleaving operation are two data streams A and B, each with its own data arrival rate. The result is an output stream C — with a new rate, different from the two source rates, determined by the formula above.

We denote the operation with the symbol #.

We define the de-interleaving operation by means of two operations.

1. Left-hand de-interleaving, producing stream A in the form:

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

2. Right-hand de-interleaving, producing stream B in the form:

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

We denote de-interleaving operations 1 and 2 with the symbols & and %.

The argument of the de-interleaving operation is an interleaved data stream together with a rational number specifying the arrival rate of the stream being extracted. The result of the operation is a data stream with the rate determined by the formula above.

The interleaving and de-interleaving operations are complementary. This means they resemble multiplication and division on the set of natural numbers. Multiplication yields a single result, whereas division sometimes leaves a remainder; what matters is also what we divide by, and in what order.

I defined the sum operation as follows:

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

The faster stream dictates the rate of the result: each of its elements is joined (the symbol | denotes tuple concatenation) with the element occupying the co-indexed slot of the slower stream.

The difference, on the other hand, is described by the formula:

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

We denote these operations with the symbols + and -.

We formally define the sequence shift operation as follows: for a stream S = (sn, Δ) and m ∈ ℕ

$$\tau_m(S) := \left( (s_{n+m})_{n=0}^{\infty},\ \Delta \right)$$

i.e. the stream shifted by m samples, that is, by a time of m·Δ. Operationally, this means shifting access to the data by a given number of intervals between consecutive elements. For instance, for data arriving once per second from the source stream, a shift operation of 3 shifts the result by 3 seconds.

I denote the shift operation with the symbol >.

The last operation within the defined algebra is the aggregation and serialization operation — abbreviated as Agse. Although it may look like two separate operations, I have defined a two-argument operator implementing the logic of a sliding data window. The first argument is the window's hop, the second is its width. The hop is a natural number specifying by how much the sliding data window must be shifted over the stream. We assume that the source data stream is split with respect to the data schema, which modifies its arrival rate. The window width is an integer, non-zero. Negative width values reverse the order in which the resulting elements are created, mirror-image fashion. Positive values preserve the sequential character of the sliding data windows created.

I denote the Agse operation with the symbol @.

To summarize, the algebra underlying the declarative query language is as follows:

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

where the first element of the pair defining the algebra is the data model (s_n — the data series, Δ_s — its regular time interval), and the second is the set of operations formally defined on this data model.

## 2.3. Formal Foundations and Proofs

In the chapter on the algebra of regular time series I presented a set of operators together with the equations describing them. I deliberately omitted formal proofs there — I wanted to first show *what* the system does before explaining *why* it is allowed to do so. This page fills that gap. Here I gather the formal skeleton of the algebra: the connection between the stream operators and covering-system theory, along with proofs of the theorems underlying the correctness and optimization of query plans.

**Info**

The entire construction below stays within a single domain — the rational numbers. This is not a stylistic choice. It is the whole point. Beatty's theorem needs irrational numbers, which a computer does not have. Fraenkel's theorem lets us descend to rational numbers. The proofs on this page show that the inter-

leaving and de-interleaving operations are a special case of Beatty sequences satisfying Fraenkel's postulates — and are therefore realizable using rational numbers alone.

### Covering systems as the foundation

The literature on covering systems [4] belongs to combinatorics and cryptanalysis within number theory. The problem under consideration is how to determine a partition of the set of positive natural numbers. We say that two sequences partition the set of positive natural numbers if the sets formed from the elements of these sequences have an empty intersection, and their union forms the set of positive natural numbers.

The basis for these considerations is the parameterized Beatty sequence. In its general form it is written using the floor function:

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

This single definition generates an entire family of sequences. Partition results always concern a **pair** of its instances with different parameters: we write the pair as B(α, α′) and B(β, β′), where the second notation denotes the complementary term.

The parameters of this sequence have a clear geometric interpretation:

- α denotes the density of the sequence,
- 1/α denotes the slope,
- α′ denotes the offset,
- −α′/α denotes the y-intercept (the point where it crosses the y-axis).

The Beatty theorem guarantees a partition of the set for irrational numbers. The Fraenkel theorem is a generalization that — crucially for us — also allows rational numbers, provided five postulates are satisfied (quoted in the introductory chapter). An accessible proof of Fraenkel's theorem can be found in K. O'Bryant's paper *"Fraenkel's partition and Brown's decomposition"* [23].

The remainder of this page boils down to a single idea: showing that the stream operators are, in essence, machines generating Beatty sequences that partition (cover) the set of natural numbers.

### Tools: floor and ceiling properties

The proofs rely almost exclusively on the floor function (⌊x⌋ — the integer part) and the ceiling function (⌈x⌉ — the smallest integer not less than x). I therefore first present a set of identities that will be used repeatedly. Let $x \in \mathbb{R}$, and let C denote an integer:

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

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

$$\lfloor x + C \rfloor = \lfloor x \rfloor + C$$

(the last identity holds for every C ∈ ℤ). Additionally, in analyzing the residue of the de-interleaving sequence we will use relationships tying the greatest common divisor (gcd) to the domain of the quotient a/b. For a, b ∈ ℕ>0:

$$\gcd(a, b) = b \iff \frac{a}{b} \in \mathbb{N}$$

and otherwise:

$$1 \leq \gcd(a, b) \leq \min(a, b)$$

These two cases disjointly cover the entire domain of interest to us — which will let us carry out a proof "by cases."

## Operators in formal notation

The operators introduced in the query language have their formal counterparts. The table below ties the formal notation (used in the proofs) to the symbols found in the query language:

| Operation | Formal symbol | Symbol in the query language |
|---|---|---|
| Projection | π | field list after `SELECT` |
| Selection | σ | logical condition |
| Sum | Σ | `+` |
| Difference | δ | `-` |
| Interleaving | φ | `#` |
| De-interleaving and its complement | Θ, ~Θ | `&` , `%` |
| Aggregation and serialization (AGSE) | Ψ | `@` |
| Shift | τ | `>` |

For the proofs to be self-contained, I restate two definitions I will refer to directly.

**Interleaving** φ(A, B) produces an output stream whose successive tuples are determined by the rule:

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

**De-interleaving** is defined by two complementary formulas — operator Θ, which recovers the original stream, and operator ~Θ, which determines the "remainder" of the de-interleaving:

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

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

### Theorem 1: interleaving guarantees set coverage

**Note**

**Theorem.** The interleaving operation guarantees sequential coverage of both index sets of the data streams that are its arguments: every element of stream A and every element of stream B is selected exactly once, in order, without gaps and without repetition.

**Proof.** Since 0 < z < 1, the increment

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

equals 0 or 1 for every n ≥ 0. The interleaving equation selects an element of stream B exactly at those steps where dn = 0 (the equality branch), and an element of stream A exactly at steps where dn = 1.

Consider the selection index for sequence B: xn = n − ⌊nz⌋. In a single step, xn+1 − xn = 1 − dn: the index increases by exactly 1 at every step selecting from B, and otherwise remains unchanged. So if n < n′ are two consecutive steps selecting from B, then xn′ = xn + 1. The first step selecting from B is n = 0, since 0 < z < 1 implies ⌊0⌋ = ⌊z⌋ = 0, i.e. d0 = 0, and x0 = 0. Selections from sequence B therefore use indices 0, 1, 2, … in order, without gaps or repetitions.

Symmetrically: the selection index for sequence A, i.e. ⌊nz⌋, increases by exactly 1 at every step selecting from A (dn = 1), and otherwise remains unchanged; at the first such step its value is 0 (all earlier steps have d = 0). The elements of sequence A are therefore also selected exactly once each, in order. □

### Theorem 2: de-interleaving satisfies Fraenkel's postulates

This is the central theorem of this page. It proves that the two sequences describing the de-interleaving operation are a special case of Beatty sequences satisfying the postulates of Fraenkel's theorem for rational numbers. Without this theorem, the whole system remains merely a promise.

**Note**

**Theorem.** Let a, b ∈ ℕ>0 represent the rational ratio of the rates of the component streams, Δa/Δb = a/b. Both tuple-selection sequences describing the de-interleaving operation are — up to the index alignment shown in the proof —

a special case of Beatty sequences satisfying the postulates of Fraenkel's theorem for rational parameters. Consequently they partition the set $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$, i.e. the index set of the interleaved stream, and de-interleaving exactly inverts interleaving using rational-number arithmetic alone.

**Proof — part one (reduction to Beatty form).** The tuple-selection sequence for the de-interleaving residue (operator ~Θ) has the form:

$$\left(n + \left\lfloor \frac{nb}{a} \right\rfloor\right)_{n=0}^{\infty}$$

Its initial term (n = 0) equals 0; the terms for n ≥ 1 form the Beatty part. For n ∈ ℕ, by the property ⌊x + C⌋ = ⌊x⌋ + C, we have n + ⌊nb/a⌋ = ⌊n + nb/a⌋, so we seek α, α′ such that:

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

Reading off the slope and the offset: with a shift of α′ = 0 we obtain α = a/(a+b), and the selection sequence restricted to n ≥ 1 is exactly:

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

**Proof — part two (verifying the five postulates and determining the residue).** We check the postulates of Fraenkel's theorem in turn for α = a/(a+b), α′ = 0:

1. The value α = a/(a+b) for a, b > 0 is greater than zero and less than one.
2. The condition α + β = 1 is satisfied for β = b/(a+b).
3. For α′ = 0 the postulate is equivalent to postulate 1.
4. The postulate is vacuous, since α is a rational number.
5. The smallest number q for which qα ∈ ℕ is q = (a+b)/gcd(a,b); then the condition 1/q ≤ α + α′ = α is satisfied, and the condition ⌈qα′⌉ + ⌈qβ′⌉ = 1 with α′ = 0 forces ⌈qβ′⌉ = 1, i.e. 0 < β′ ≤ gcd(a,b)/(a+b). Every admissible value generates the same sequence (the complement of the sequence B(a/(a+b), 0) in ℕ is unique); we take β′ = gcd(a,b)/(a+b).

The sequence complementing B(a/(a+b), 0) in the sense of Fraenkel's postulates is therefore:

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

After reindexing n ↦ n + 1, so that it runs from n = 0 — matching the selection sequences in the definition of de-interleaving — it takes the form:

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

Expanding the above expression:

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

Comparing this — term by term for n ≥ 0 — with the tuple-selection sequence of the recovered stream (operator Θ):

$$\left( n + \left\lceil \frac{(n+1)a}{b} \right\rceil \right)_{n=0}^{\infty}$$

and factoring out the integer part n + 1 via the property ⌊x + C⌋ = ⌊x⌋ + C, the claim reduces (after substituting n for n + 1, so that n ranges over ℕ>0) to the identity:

$$\left\lfloor n\frac{a}{b} - \frac{\gcd(a,b)}{b} \right\rfloor + 1 = \left\lceil n\frac{a}{b} \right\rceil, \quad n \in \mathbb{N}_{>0}$$

**Proof — part three (case analysis).** Using the properties of the coefficient gcd(a, b), we consider two disjoint cases covering the entire domain.

*Case 1: gcd(a, b) = b, i.e. a/b ∈ □.* Then n·a/b ∈ ℕ, so by the identity ⌊x⌋ = ⌈x⌉ ⟺ x ∈ ℤ we have ⌈n·a/b⌉ = ⌊n·a/b⌋, and by ⌊x + C⌋ = ⌊x⌋ + C:

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

Both sides of the identity being proved coincide.

*Case 2: b ∤ a, i.e. 1 ≤ gcd(a, b) < b and 0 < gcd(a,b)/b < 1.*

If n·a/b ∉ ℤ, then by ⌊x⌋ + 1 = ⌈x⌉ ⟺ x ∈ ℝ □ ℤ we have ⌈n·a/b⌉ = ⌊n·a/b⌋ + 1. The fractional part of n·a/b is a nonzero multiple of gcd(a,b)/b, and hence at least gcd(a,b)/b; subtracting gcd(a,b)/b from n·a/b therefore cannot drop below the integer ⌊n·a/b⌋, whence:

$$\left\lfloor n\frac{a}{b} - \frac{\gcd(a,b)}{b} \right\rfloor = \left\lfloor n\frac{a}{b} \right\rfloor$$

and the identity being proved holds.

If n·a/b ∈ ℤ, then ⌈n·a/b⌉ = n·a/b, and since 0 < gcd(a,b)/b < 1:

$$\left\lfloor n\frac{a}{b} - \frac{\gcd(a,b)}{b} \right\rfloor = n\frac{a}{b} - 1$$

which again gives the identity being proved.

Both selection sequences describing the de-interleaving operation are therefore — up to the unit reindexing from part two — Beatty sequences satisfying Fraenkel's postulates for rational parameters: the pair B(a/(a+b), 0) and B(b/(a+b), gcd(a,b)/(a+b)) partitions the set ℕ, and together with the initial residue term 0 from part one — the set ℕ₀, the full index set of the interleaved stream. The recovered stream and the residue are therefore exact. □

**Note**

**Corollary (exact invertibility over the rationals).** For streams with rational rates, the operators Θ and ~Θ recover the component streams of φ(A, B) exactly (bit for bit): no tuple is lost, duplicated, or reordered relative to its component stream. The pair (φ; Θ, ~Θ) therefore behaves like multiplication and division, and the pair (Σ; δ) like addition and subtraction, on the set of regular time series.

**Warning**

The practical takeaway from this proof: the implementation must never leave the domain of rational numbers, not even momentarily. An implicit cast of an intermediate result to a floating-point number breaks the assumptions of the theorem above. Materialization into floating-point form must be deferred until an explicit application of the floor or ceiling operation.

## Operator properties used in optimization

Based on the algebra presented, a number of properties of data streams can be shown. They have direct application in the data-management system — during query-plan optimization and result interpretation.

### Disruption of event ordering

**Note**

**Theorem.** The order of elements in a stream does not reflect the actual order in which the elements occurred in the real world.

**Proof (by counterexample).** Consider two streams:

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

The expression φ(Epsilon, Alpha) produces the output stream:

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

In stream Tau, the tuple labeled c occurs after the tuple labeled 5. Yet tuple c appears in stream Epsilon at second 9, while tuple 5 appears in stream Alpha at second 10. The natural order of events has been violated in the resulting stream. Conclusion: when analyzing time embedded in streams, applying the de-interleaving operation is necessary to recover the original form of the data streams. □

### Commutativity of summation

**Note**

**Theorem.** The stream-summation operation, disregarding attribute order, is commutative.

**Proof.** Assume Δa ≤ Δb; the opposite case is symmetric. The first case of the sum definition gives, as the n-th element of stream Σ(A, B), the tuple:

$$c_n = \left(a_n,\ b_{\lfloor n\Delta_a/\Delta_b \rfloor}\right)$$

whereas for Σ(B, A) the roles of the arguments are swapped, and its second case applies (or, at Δa = Δb, its first), giving as the n-th element:

$$c_n = \left(b_{\lfloor n\Delta_a/\Delta_b \rfloor},\ a_n\right)$$

Both streams carry Δc = Δa. They therefore coincide up to the order of the joined attributes. □

### Interleaving alignment method

The interleaving operation is not commutative in general: since 0 < z < 1, at n = 0 the equality branch of the interleaving definition always applies, so the stream φ(A, B) begins with element $b_0$, while the stream φ(B, A) begins with element $a_0$. Interleaving is, however, equivariant with respect to time shifts matched to the streams' rates — which is valuable for query-plan optimization.

**Note**

**Theorem.** If numbers i, k ∈ ℕ are chosen such that i·Δa = k·Δb (both arguments shifted by the same amount of time), then interleaving the shifted streams equals the interleaving of the original streams shifted by the sum of these numbers.

Formally:

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

**Proof.** Both sides carry the interval Δc from the definition of interleaving, so it suffices to compare the elements. From the assumption i·Δa = k·Δb:

$$(i+k)z = (i+k)\frac{\Delta_b}{\Delta_a+\Delta_b} = \frac{i\Delta_b + i\Delta_a}{\Delta_a+\Delta_b} = i \in \mathbb{N}$$

and hence, by the property ⌊x + C⌋ = ⌊x⌋ + C, for every n ≥ 0 and m := n + i + k:

$$\lfloor mz \rfloor = \lfloor nz + i \rfloor = \lfloor nz \rfloor + i$$

In particular ⌊mz⌋ = ⌊(m+1)z⌋ if and only if ⌊nz⌋ = ⌊(n+1)z⌋: position n on the left-hand side of the claim and position m of stream φ(A, B) select the same branch of the interleaving definition. On the equality branch the right-hand side selects the element:

$$b_{m-\lfloor mz \rfloor} = b_{n+i+k-\lfloor nz \rfloor - i} = b_{(n-\lfloor nz \rfloor)+k}$$

which is exactly the element (τk(B))n−⌊nz⌋ selected by the left-hand side; on the inequality branch it selects:

$$a_{\lfloor mz \rfloor} = a_{\lfloor nz \rfloor + i}$$

which is the element (τi(A))⌊nz⌋ — again matching the left-hand side. Both streams coincide element by element. □

## Why this matters

The theorems presented are not formalism for its own sake. Each of them plays a concrete role in the working system:

- **Theorems 1 and 2** guarantee that the pairs of operations interleaving/de-interleaving and sum/difference are complementary — data is neither lost nor duplicated in an uncontrolled way. They are what allow us to treat these operations like multiplication/division and addition/subtraction on the set of regular time series.
- **Theorem 2** in particular proves that the entire construction can be realized using rational numbers alone — and thus deterministically and exactly on a computer. This is the condition without which RetractorDB could not exist.
- **The theorems on operator properties** (commutativity of summation, interleaving alignment, ordering disruption) provide rewrite rules for stream expressions. The query-plan optimizer uses them to transform plans into cheaper-to-execute forms without changing the result.

The branch of mathematics in which these equations are situated is the theory of covering systems [4] within number theory. I presented the full formalism along with a complete set of proofs in the paper A Deterministic Method for Processing Data Sequences [3].

**Info**

A numerical verification of the equations above — Python prototypes operating on rational numbers (the `Fraction` library) — can be found on the Model Implementation page and in the repository github.com/michalwidera/equations.

## 2.4. Algebraic Expressions

The defined algebra entails the possibility of defining algebraic expressions. Typical algebraic expressions over the set of rational numbers are material covered in elementary school. Algebraic expressions in RetractorDB occur in two forms. In the field list of the SELECT command, we have the typical expressions familiar from school. In the argument list of the SELECT command's FROM clause, we have an algebraic expression built on the newly defined algebra.

This means that in the field list after the SELECT clause, the plus operator means one thing, while in the FROM clause it means something else entirely. An innocent-looking query from the definition combines two entirely different worlds and concepts: one, an algebra based on numbers; the other, based on regular time series.

Example. As an example, we present an algebraic expression built over the set of regular time series (hereafter called streams). Assume the existence of two streams: A(a1 int, a2 int),1 and B(b1 int),½ — where,

- A denotes a stream containing, in each record, two fields of type int — a1 and a2 — arriving once per second, and
- B contains, in each record, a field of type int named b1 arriving twice per second.

The algebraic expression C=A+B creates a data stream with fields C(a1 int, a2 int, b1 int),½.

To interleave a data stream, sets A and B should have the same data schema. Let's assume, then, that there is a stream D(d1 int),1 — arriving, like stream A, once per second.

The algebraic expression E=B#D creates the stream: E(e1 int),⅓. The rate ⅓ comes from the formula (1*½)/(1+½). You will find this formula in the definition of the interleaving operation.

For streams defined this way, the following expression is still valid:

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

And such expressions may appear as valid, with respect to the developed time-series algebra, in the body of a query.

### Further examples

Continuing with the streams defined above, A(a1 int, a2 int),1, B(b1 int),½ and D(d1 int),1, and the output streams C=A+B and E=B#D — below is a further set of valid algebraic expressions. Equivalents of all these expressions appear in FROM clauses

of queries in the system's integration tests, and are verified on every build of the project.

Interleaving the result of an interleave:

```
G=E#D
```

Stream E has rate ⅓, stream D has rate 1, both share the same schema with a single int field. The interleaving formula gives a rate of (⅓·1)/(⅓+1)=¼, so G(g1 int),¼. The result of one operation is a fully valid stream and can be the argument of the next one.

Sum of three streams:

```
H=A+B+D
```

Summation joins tuples, so the result schema is the concatenation of the schemas, and the rate is dictated by the fastest component: H(a1 int, a2 int, b1 int, d1 int),½.

Sum with a shifted component, and a shift of an interleaving argument:

```
I=D+((A+B)>1)
J=(B>1)#D
```

Shifting a sequence does not change the stream's rate — it only changes data access by a given number of samples. Therefore I has rate min(1,½)=½, and J — just like E — has rate ⅓.

De-interleaving:

```
K=E&1
L=E%½
```

The right-hand argument of the de-interleaving operators is a rational number, not a stream. Substituting into the de-interleaving formulas: K has rate (⅓·1)/|⅓−1|=½ — left-hand de-interleaving recovers stream B from the interleave E. Similarly L has rate (⅓·½)/|⅓−½|=1 — right-hand de-interleaving recovers stream D. De-interleaving is the inverse of interleaving, just as division is the inverse of multiplication.

Difference:

```
M=C-1
```

Difference is the inverse operation to sum — it extracts, from the joined stream C, the component indicated by the rational number on the right-hand side of the operator.

Aggregation and serialization:

```
N=A@(1,4)
P=A@(1,-4)
R=A@(2,2)
S=(A@(2,2))@(1,1)
```

N creates a sliding window of width 4 shifted by one element, P — thanks to its negative width — builds the same windows mirror-imaged, R creates disjoint windows (hop equal to the width). Expression S shows that the result of an Agse operation can be the argument of another Agse operation.

All of the above forms can be combined into arbitrarily complex expressions — like F=((B#D)+A)>2 from the example above — as long as the data schemas of the arguments satisfy the requirements of the respective operations.

### Coverage of examples in integration tests

Each of the expression forms cited has a counterpart in the RetractorDB repository's integration tests (directories `test/IntegrationTest_serial` and `test/IntegrationTest_parallel`), executed on every project build:

| Expression from this chapter | Form in the test | Integration test |
|---|---|---|
| C=A+B (sum) | `s1+s2`, `core0+core1` | `IntegrationTest_serial/issue167_dedu`<br>`IntegrationTest_serial/Data` (all-operators) |
| E=B#D (interleaving) | `core0#core1` | `IntegrationTest_serial/operations`, `IntegrationTest_serial/Data` (all-operators) |
| G=E#D (cascaded interleaving) | `(s1#s2)#s3`, `s1#s2#s3` | `IntegrationTest_serial/issue167_tria` |
| H=A+B+D (multi-argument sum) | `s1+s2+s3`, `s1+s2+s3+s4` | `IntegrationTest_serial/issue167_tria` |
| I=D+((A+B)>1) | `s3+((s1+s2)>1)` | `IntegrationTest_serial/issue167_dedu` |
| J=(B>1)#D and (B#D)>1 | `(core1>1)#core2`, `(core1#core2)>1` | `IntegrationTest_parallel/subquery` |
| K=E&1, L=E%½ (de-interleaving) | `core0&1.5`, `core0%4` | `IntegrationTest_serial/Data` (all-operators) |
| M=C−1 (difference) | `core0-1/2` | `IntegrationTest_serial/Data` (all-operators) |
| shift of a sum, as in F | `(s1+s2)>1`, `(core0+core1)>5` | `IntegrationTest_serial/issue167_dedu`<br>`IntegrationTest_serial/issue56_times` |
| N=A@(1,4), P=A@(1,−4), R=A@(2,2) | `core1@(1,4)`, `core1@(1,-4)`, `core1@(2,2)` | `IntegrationTest_serial/agse1` (further hop/width variants: `agse2`, `agse3`) |
| S=(A@(2,2))@(1,1) (cascaded Agse) | `signalText3@(1,1)` | `IntegrationTest_serial/agse1` |

The tests compare query execution results against pattern files, so the expressions above are verified not only syntactically, but also in terms of the values and rates of the resulting streams.

## 2.5. Model Implementation

The developed algebra equations were first implemented in Python. This is, to my knowledge, the most effective way to model and numerically verify hypotheses. Each operator was implemented inside a separate function. Operations are carried out on

rational variables (the Fraction library). Results are presented as bounded arrays. In the final implementation, however, these operators operate on infinite data structures.

### Interleaving operation

Let's start by building the interleaving operation:

**Source code**

```
## Interleaving (hash) operation on two lists with given steps (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()
```

---

**Run output**

```
$ 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
```

Running the code prints the input data A and B, along with the results of the operations A#B and B#A. As you can see, the interleaving operation is not commutative.

### De-interleaving operation

The de-interleaving operation requires implementing two complementary operations.

**Source code - even**

```
## De-interleaving (dehash) operation, 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()
```

---

**result - 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
```

---

**Source code - odd**

```
## De-interleaving (dehash) operation, 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):

  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()
```

---

**result - 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
```

This code first joins two streams and then extracts the source data back out.

### Sum operation

Summation joins two data streams arriving at different rates.

### Source code

```
## Summation operation on two lists with given steps (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()
```

---

### result

```
$  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
```

### Difference operation

The complementary operation to sum is the difference operation.

**Source code**

```
## Difference (diff) operation on two lists with given steps (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()
```

---

**result**

```
$ 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
```

**Source code**

The source code for the examples shown here can be found in the project repository under the /examples/python-model/ directory.

A JavaScript implementation can be tested directly on the site:

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

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

## 2.6. Graphical Representation

Fig. 2 shows, schematically, the relationships between the developed time-series algebra operators. In the figure, I have combined the relationships between the developed operators, their symbolic notation used in the query language, and the directions of data processing.

The figure shown is also a graphical summary of the content presented in this chapter. I hope this graphical form of representation makes it easier to absorb the rules governing the introduced operators. For clarity, the aggregation/serialization and time-shift operators have been omitted. Be aware that the diagram is incomplete without them.

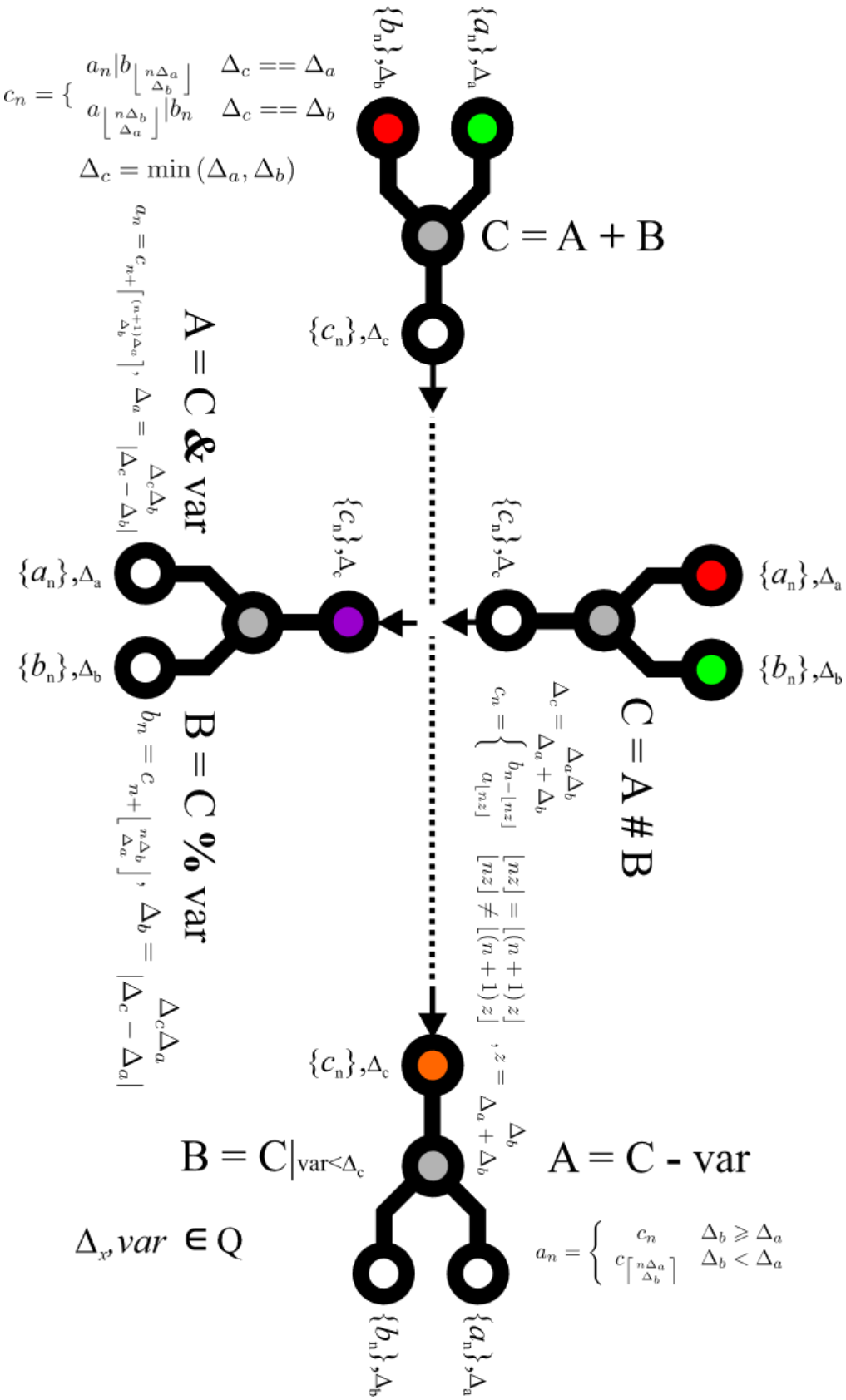



Fig. 2. Relationships between the algebra operators

## 2.7. Summary

I initially modeled these equations as Python programs. The formal form presented here took shape only at the very end of the search process. To numerically prove the correctness of the developed equations, I constructed sequences of operations on streams. If any elements got lost in the course of carrying out the operations shown, it meant I had made a mistake. It turns out, for instance, that it is essential in the implementation never to leave the domain of rational numbers, even for a moment. A mistake can be made by accident, by implicitly casting the result to a floating-point number. Materializing the result as a floating-point number must be deferred, in the calculations, until the result is explicitly carried over via the floor or ceiling operation. If we assemble a Python program into a sequence of operations on infinite streams and no data disappears as a result of that operation — we have an object ready for further research and formal analysis, ready for a formal mathematical proof of correctness. The formal proof (the mathematical formalism) can be found in the paper titled A Deterministic Method for Processing Data Sequences [3].

The branch of mathematics that contains the research related to these equations is called covering systems [4] within number theory.

**Info**

Presenting the mathematical foundations of the system is necessary in order to understand the further technical aspects of the solution. The methods presented go beyond the standard material currently taught in technical-science degree programs. This is because I drew the mathematical foundations from an area that, to my knowledge, had not previously been applied in engineering. These are methods that make it possible to build a new way of processing data. This is one of the aspects that sets RetractorDB apart from other similar solutions.

# Chapter 3

# Query Language Construction

Communication between the developed system and the user takes place through a purpose-built, declarative query language. The language's construction is based on the algebra presented in the previous chapter. Just as, for relational systems, relational algebra forms the basis for the SQL language — in our case, the developed algebra forms the basis for the RQL query language.

RQL stands for *RetractorDB Query Language*. Its syntax is very similar to SQL syntax. Bear in mind, however, that the correct term here is a *False Friend*. That is, it looks like SQL, but has little in common with it.

Valid statements in RQL currently begin with a handful of keywords. The most recognizable is the command starting with the keyword SELECT, followed by a list of attributes in the form of algebraic expressions — an algebra based on real numbers.

Statements are written in a text file. Its extension is conventionally .rql, but any other extension will also be accepted and processed. An RQL text file contains a sequence of statements beginning with defined keywords. Comments are prefixed with the # character.

The query language was implemented using the Antlr4 parser generator [5]. The RQL grammar is written down, defined, and after every modification is compiled into the language in which RetractorDB itself was built. Every statement in a query file that is not a comment is compiled, processed, and modifies the system's internal state. A statement may span multiple lines — line continuation is signaled with a \ character at the end of the line.

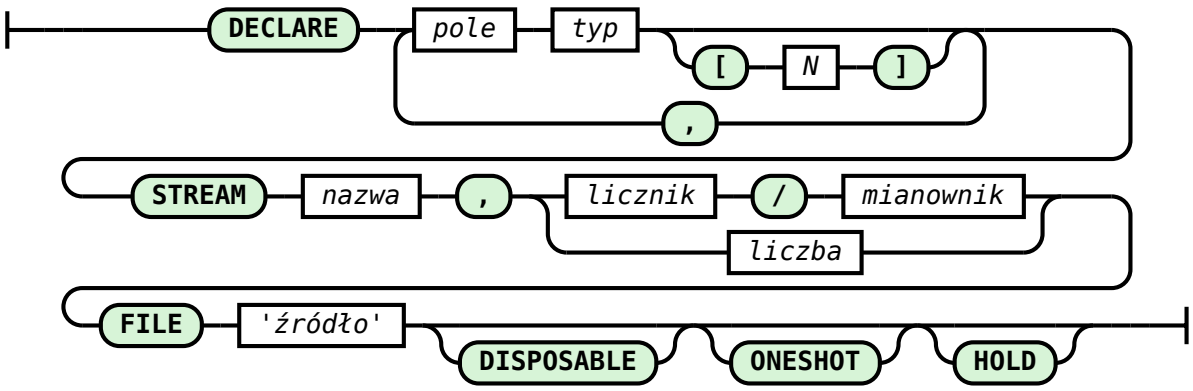
## 3.1. DECLARE Command

The DECLARE command is used to declare a data source.

Its syntax is described as follows:

```
DECLARE field type[N] [, field type[N]]
STREAM name, rate
```

```
FILE source
[DISPOSABLE]
[ONESHOT]
[HOLD]
```




DECLARE command syntax diagram

*Fig. 3. DECLARE command syntax diagram*

The railroad diagram in Fig. 3 was generated from the `declare_statement` rule in the system's ANTLR4 grammar (`RQL.g4`). The diagram is read by following the lines from left to right: rounded green boxes are keywords and symbols entered literally, rectangles are values supplied by the user. A loop looping back through a comma means multiple field declarations are possible; the branch at the rate shows it can be written as a fraction (numerator/denominator) or as a single number; tracks bypassing DISPOSABLE, ONESHOT, and HOLD mean each of these directives is optional.

## Field types

Every field has a name and a type. Available types:

| Type | Size | Description |
|---|---|---|
| `BYTE` | 1 B | unsigned 8-bit integer |
| `INTEGER` | 4 B | signed 32-bit integer |
| `UINT` | 4 B | unsigned 32-bit integer |
| `FLOAT` | 4 B | 32-bit floating-point number |
| `DOUBLE` | 8 B | 64-bit floating-point number |
| `STRING` | N B | fixed-length byte string of length N |

### Field arrays (`type[N]`)

Any field can be given an array multiplier `[N]` — the field then occupies `N × type_size` bytes and creates `N` consecutive positions in the record schema:

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

The field `coef INTEGER[25]` creates a record of size $25 \times 4 = 100$ bytes and gives access to indices `filter[0]` ... `filter[24]`. This is the standard way of passing coefficient arrays (e.g. FIR filters) into the system.

Multiple fields of different types can be combined in a single record:

```
DECLARE id UINT, value FLOAT, name STRING[16] \
STREAM measurement, 0.1 \
FILE 'sensor.dat'
```

Record size: $4 + 4 + 16 = 24$ bytes.

RetractorDB, running under Linux, reads and writes data to files. On Linux, access to most resources is carried out through access to various kinds of files. This approach unifies the way data is accessed.

An example of a command that creates an object in RetractorDB returning random values from the /dev/random stream 10 times per second, with values of type int, looks as follows:

```
DECLARE random_field INTEGER \
STREAM random_stream, 0.1 \
FILE '/dev/random'
```

The source mentioned in the command, if declared as a text file with the .txt extension, will be interpreted by the system as a continuous, unbounded data file read line by line. Upon reaching the end of the file, reading resumes from the beginning. This functionality is built into RetractorDB. Basic support for the format is provided — if we specify two integer fields in the declaration, and the file contains two integer values separated by a space, those values will be read as consecutive elements of the record.

```
DECLARE field_1 INTEGER \
STREAM cyclic_stream, 0.1 \
FILE 'file.txt'
```

For the parsing of the file to happen automatically, the file must have the .txt extension. At present this behavior is hard-coded and not configurable. I plan to change this in the future.

**Note**

The functionality described here is covered by the test: `Pattern7`, described in the appendix Integration Tests.

If the input data file has the .dat extension, it will be treated as a binary file, and reading from it will likewise loop. Looping means that after the last value in the source file has been read, the read position is moved back to the beginning. Data from such a file is read in an infinite loop, returning to the start once it finishes.

The three optional directives (`ONESHOT`, `DISPOSABLE`, `HOLD`) control the data source's lifecycle — a detailed description and comparison table can be found in the chapter Read Options.

**Info**

Support for NULL values (per field) is implemented in RetractorDB. Null metadata is stored in the `.meta` file alongside the binary data, managed by the `metaDataStream` class.

## DECLARE Read Options

The `DECLARE` command accepts three optional directives that affect how the declared source is read and its lifecycle:

```
DECLARE field type STREAM name, rate FILE source
    [DISPOSABLE]
    [ONESHOT]
    [HOLD]
```

### ONESHOT

Without `ONESHOT`, a data source is read in an infinite loop — once the end of the file is reached, the read position returns to the beginning. `ONESHOT` disables the loop: the file is read exactly once, and once exhausted the stream returns zero or empty values.

```
DECLARE measurement INTEGER STREAM burst, 0.1 FILE 'data.dat' ONESHOT
```

Use case: one-off loading of historical data into the system.

### DISPOSABLE

Once data transfer from the source finishes, the system deletes the data file, the descriptor file (`.desc`), and the metadata file (`.meta`). The directive acts on destruction of the `storage` object.

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

`DISPOSABLE` is used together with `ONESHOT` — data is read once, then deleted after reading. This combination is useful for temporary input data files.

### HOLD

The declared source does not start reading immediately after the system starts. Physical data reading begins only when the first query requiring data from this stream arrives (e.g. an Ad Hoc query). Until the stream is queried, the system shows zero or empty values for it.

```
DECLARE sparse INTEGER STREAM optional_stream, 1.0 FILE 'sparse.dat' HOLD
```

Use case: data sources activated conditionally, e.g. on user request via `xqry`.

**Comparison table**

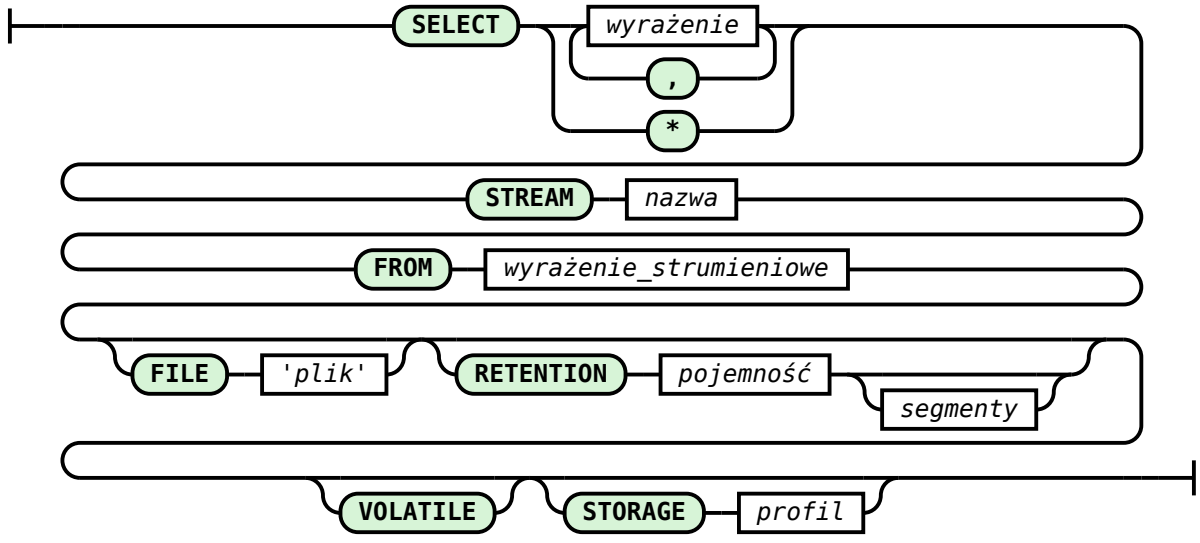
| Directive | Read loop | Deletes files after reading | Delayed read start |
|---|---|---|---|
| *(default)* | yes | no | no |
| `ONESHOT` | no | no | no |
| `DISPOSABLE` | yes | yes | no |
| `HOLD` | yes | no | yes |

## 3.2. SELECT Command

Every SELECT command in RetractorDB creates a continuous query. These queries run from the moment they appear in the system until the system shuts down.

The syntax of the SELECT command is as follows:

```
SELECT algebraic_expression [, algebraic_expression]
STREAM output_stream_name
FROM stream_algebraic_expression
[FILE 'artifact_file_name']
[RETENTION capacity [segments]]
[VOLATILE]
[STORAGE profile]
```




SELECT command syntax diagram

*Fig. 4. SELECT command syntax diagram*

The railroad diagram in Fig. 4 was generated from the `select_statement` rule in the system's ANTLR4 grammar (`RQL.g4`). The diagram is read by following the lines from left to right: rounded green boxes are keywords and symbols entered literally, rectangles are values supplied by the user. The branch after the word SELECT shows that the field list is either an asterisk (the full record) or one or more expressions separated by commas (a loop looping back through a comma). Tracks bypassing the FILE, RETENTION (with an optional second parameter — the number of segments), VOLATILE, and STORAGE clauses mean each of them is optional.

Readers familiar with SQL will immediately notice that the command shown above differs significantly from what they know from relational databases.

The first difference, beyond syntax, is that once entered into the system, these commands run until the system shuts down. Every SELECT command is a continuous query. The STREAM clause requires the author to give every query a unique name. While the algebraic expressions in the SELECT clause's field list don't differ from the form familiar from relational systems, the stream algebraic expression must satisfy the conditions presented in the previous chapter on algebraic expressions. The optional FILE and RETENTION clauses provide processes for directing results and managing their retention. Old, segmented output files can be deleted on an ongoing basis, keeping room in the system for new data in continuous motion.

An example of a query creating a new data stream might be the following RQL command.

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

A query built this way assumes that someone has declared streams A and B. This could have been done with the DECLARE keyword or with another SELECT command. Based solely on the line containing the query, we cannot tell how fast the data of stream str1 arrives. This information is computed at compile time, based on streams A and B and the algebraic expression in the FROM clause.

**Note**

The functionality described here is covered by the tests: `simple`, `Pattern2`, described in the appendix Integration Tests.

The VOLATILE clause creates an ephemeral form of the query. A query with this clause holds only a single record in memory — only the descriptor describing the data structure appears on disk.

The STORAGE clause allows choosing how the artifacts created are managed and stored. The full table of types, with a description of each, is in the chapter Storage Types.

### FROM clause operators

The stream algebraic expression in the `FROM` clause can include:

| Operator | Syntax | Description |
|---|---|---|
| Sum | `A + B` | Joins two streams — see Summation Sequencing |
| Interleaving | `A # B` | Interleaving of two streams — see Interleaving Sequencing |
| Shift | `A > N` | Shifts the read window by N samples |
| AGSE window | `A @ (k, w)` | Sliding data window — see AGSE Sliding Data Window |
| Aggregate | `A.min` / `A.max` / `A.avg` / `A.sumc` | Reduces a multi-field record to a single value — see Aggregate Operators |

**Note**

The shift operator `A > N` is covered by the test: `issue56_timeshift`, described in the appendix Integration Tests.

**Note**

Null-value propagation through SELECT expressions is covered by the test: `issue121_null_propagation`, described in the appendix Integration Tests.

### Summation Operation Sequencing

Data flows into the system and is processed within it. The order in which it arrives and is processed can be described by the term sequencing. The way data is combined is described by the algebraic expression placed in the FROM clause. These expressions are written as a series of algebraic operations subject to strict rules — rules similar to those we learned in elementary school, governing arithmetic operations on numbers such as addition, multiplication, division, and subtraction.

Let's start by analyzing the following query:

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

I'll save the query in a file named qplan1.rql. Then I'll run the following commands:

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

The swirly program was installed from its GitHub repository [6]. This program is used to generate marble diagrams used to explain the behavior of RxJs asynchronous operations [7].

The modification I applied for my use case is an alternative meaning for the vertical lines. In my case, vertical lines separate uniform time intervals — showing the number of cycles requested at invocation time (in this case, 3 cycles). The generated image is shown in Fig. 5:

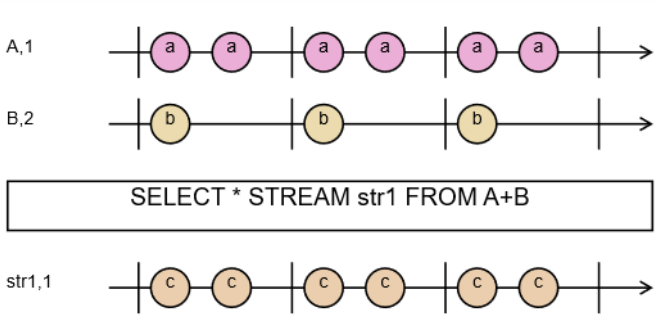



Fig. 5. Marble diagram — the sum operation

A few words of explanation are needed here about this generator and how its input is produced. I built into the compiler an option for visualizing the execution of a sequence of operations. Diagrams produced by the Swirly program are one convenient way of presenting time dependencies. On input, the Swirly program expects a text file describing the diagram. A generator that simulates the requested number of cycles and builds a file for Swirly has been built into the compiler.

When xretractor is given, as its first parameter, the name of the file containing the query execution plan, it requires a second parameter ( -w [–diagram] ) — indicating that we expect marble-diagram output. The required argument of the -w parameter is two numbers separated by a colon. The first tells the program whether to insert time separators into the diagram (the vertical lines separating cycles); the second parameter is how many cycles should be shown in the diagram.

If you look at the generated out.txt file, you'll see the following content:

```
% 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
```

In this file, note the data shown in the comments. These are the times computed while generating the diagram, relative to the scale shown in the marble diagram. As you can see, for our query the minimum window interval is 1 second and the maximum is 2 seconds. The identified and computed grid is half a second. On the diagram, each letter or dash represents a half-second period between successive operations.

We can manually change the generated content. If we replace the content as follows:

```
-|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
```

and then run the swirly program again, we'll see a more detailed picture showing the sequence of events occurring in the system.

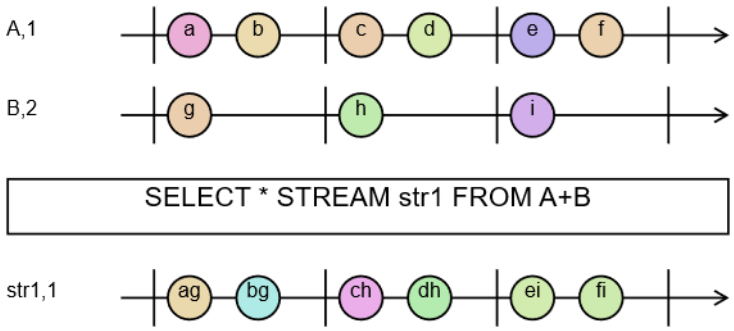



Fig. 6. Marble diagram — Sum, modified diagram

In the diagram shown in Fig. 6, you can see which marbles were joined and which marbles they were formed from. Remember, though, that this is a manually corrected image, made for the purposes of this work — the generator built into the compiler does not implement this functionality.

**Note**

The functionality described here is covered by the tests: `Pattern1`, `issue167_triarg`, described in the appendix Integration Tests.

### Interleaving Operation Sequencing

Let's now analyze the interleaving operation. Let's create a file qplan2.rql with the following content:

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

Aside from the # symbol instead of the + symbol in the from clause, the two files are identical. Let's run the compilation and the swirly program. The resulting graphic will look as follows:

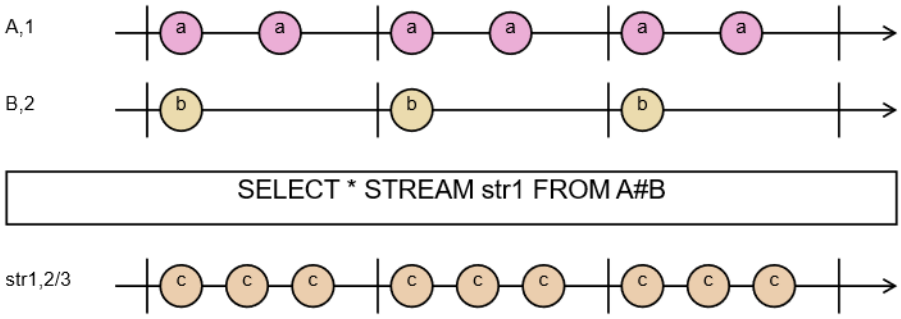



Fig. 7. Marble diagram — the interleaving operation

In Fig. 7 we see a change. The marbles of stream str1 have been evenly arranged in time. The events occurring in the declared input data streams have not changed. What has changed is the way the output stream str1 is built.

If we look at the generated text schema, we'll see that the time values have changed too:

```
% Minimum interval is 666ms
% Maximum interval is 2000ms
% Grid time is 333ms, divider:2
% Full cycle step count in grid is 6
```

I encourage further experimentation with this way of presenting the defined operations on time series.

**Note**

The functionality described here is covered by the tests: `operations`, `Pattern1`, described in the appendix Integration Tests.

### VOLATILE Clause

The `VOLATILE` clause in the `SELECT` command creates a stream that holds only a single record in memory. On disk, only the `.desc` descriptor file describing the data schema appears — the data itself is never written.

### Behavior

```
SELECT expression STREAM name FROM source VOLATILE
```

Internally, the compiler sets the storage type to `MEMORY` with a capacity of `1`:

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

This means that:

- the in-memory buffer always holds only the single, most recent record,
- data never reaches disk,
- the `.desc` descriptor is still created — other processes can learn the stream's schema.

### Difference from `STORAGE MEMORY`

| Property | `VOLATILE` | `STORAGE MEMORY` |
|---|---|---|
| Buffer capacity | always 1 record | depends on `RETENTION` |
| `RETENTION` clause | ignored | applied |
| Descriptor on disk | yes | yes |
| Data on disk | no | no |

`VOLATILE` is useful when the query result is being pulled by `xqry` on an ongoing basis and history is not needed — e.g. the current value of a sensor exposed by the operating system.

### Example

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

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

The `scaled` stream contains, at every moment, a single, current value. The `xqry` process can read it via shared memory.

## STORAGE Types

The `STORAGE` clause in the `SELECT` command, and the `SUBSTRAT` directive, accept one of the following identifiers. Each maps to a specific data-accessor class in the implementation.

### Type table

| Keyword | C++ class | Retention | Shadow | Purpose |
|---|---|---|---|---|
| `DEFAULT` | `groupFile<posixBinaryFileWithShadow>` | yes | yes | Default production mode; a `.shadow` file protects modifications |
| `DIRECT` | `groupFile<posixBinaryFile>` | yes | no | Retention without shadow protection |
| `MEMORY` | `memoryFile` | yes (RAM) | no | Data in memory only; circular buffer, never written to disk |
| `POSIX` | `posixBinaryFile` | no | no | A single binary file; no retention |
| `POSIXSHD` | `posixBinaryFileWithShadow` | no | yes | A single file with shadow protection; no retention |
| `GENERIC` | `genericBinaryFile` | no | no | Generic binary file |
| `DEVICE` | `binaryDeviceRO` | no | no | Binary device; read-only; looping depends on `ONESHOT` |
| `TEXTSOURCE` | `textSourceRO` | no | no | Text file; read-only; looping depends on `ONESHOT` |

**Retention** — artifacts are rotated, older files are deleted automatically (requires `RETENTION` on `SELECT`).
**Shadow** — every modification is written to a separate `.shadow` file; historical data is protected from being overwritten.

For `MEMORY`, retention works in memory as a circular buffer: successive appends overwrite the oldest slot (`index % capacity`). Data is not segmented into files and never reaches disk.

**Note**

The `MEMORY` type (SUBSTRAT 'memory') is covered by the tests: `issue61_tmpmem` (serial and parallel), described in the appendix Integration Tests.

### When to use which

The choice depends on the environment's requirements:

- **Production environment, critical data** → `DEFAULT` (retention + shadow)
- **Production environment, historically insignificant data** → `MEMORY` (zero disk usage, retention in RAM)
- **Development and debugging** → `DEFAULT` or `DIRECT` (data visible on disk)
- **Reading from a device or a text file** → `DEVICE` / `TEXTSOURCE` (respectively)

**Example**

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

For substrates globally — the `SUBSTRAT` directive:

```
SUBSTRAT 'memory'
```

## Aggregate Operators and Expression Functions

### Window aggregates (MIN, MAX, AVG, SUMC)

Aggregate operators operate on a stream with multiple fields — typically the output stream of the `@(k,w)` operator (a data window). They reduce all fields of the record to a single value.

**Syntax**

```
FROM stream.aggregator
```

where `aggregator` is one of:

| Keyword | Behavior |
|---|---|
| `min` / `MIN` | minimum of all fields in the record |
| `max` / `MAX` | maximum of all fields in the record |
| `avg` / `AVG` | arithmetic mean of the record's fields |
| `sumc` / `SUMC` | sum of all fields in the record |

Keywords are accepted in both lowercase and uppercase.

**Output interval** Aggregates do not change the stream's rate — the output interval is the same as the source's:

$$\Delta_{result} = \Delta_{stream}$$

**Example: moving average**

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

-- a 5-element window shifted by 1
SELECT * STREAM win5 FROM src@(1,5)

-- average of the last 5 values
SELECT win5[0] STREAM ma5 FROM win5.avg
```

The `ma5` stream contains, at every moment, the average of the five most recent `src` samples.

**Example: signal filter (sumc)** An excerpt from the signal-filter implementation example:

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

`accRow.sumc` sums all fields of the `accRow` record (products of signal samples and filter coefficients), producing the output of an FIR filter.

**Example: MIN and 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
```

**Note**

The functionality described here is covered by the tests: `simple_max`, `Pattern4`, described in the appendix Integration Tests.

---

### The to_string function

The `to_string` function converts a numeric expression to a text string of a given width. The result goes into a field of type STRING in the output stream.

**Syntax**

```
to_string(expression : width)
to_string(expression)
```

The `width` parameter (a natural number after the colon `:`) specifies the output field's width in bytes. Omitting the parameter gives a default width of 32 bytes.

**Info**

The argument separator is a colon `:`, not a comma `,`. A comma is the SELECT list separator — using a comma in `to_string(x, n)` will cause a parse error.

**Example**

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

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

The `labels` stream contains the values of `src` formatted as text in a 10-byte field.

**Concatenation with a literal** The resulting string can be joined with a string literal using the + operator:

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

Output field size: 8 (from `to_string`) + 3 (literal `_ok`) = 11 bytes.

**Use cases** `to_string` is useful when exporting to systems that accept text data (Graphite, InfluxDB via `xqry`), or when creating event labels combined with `DO DUMP` output.

**Note**

The functionality described here is covered by the tests: `issue121_isnull`, `issue128_numeric_to_string`, `issue128_string_to_numeric`, described in the appendix Integration Tests.

## 3.3. RULE Command

This command is one of the most recent extensions I have developed for the system. It extends the system's functionality with an alerting mechanism.

**Note**

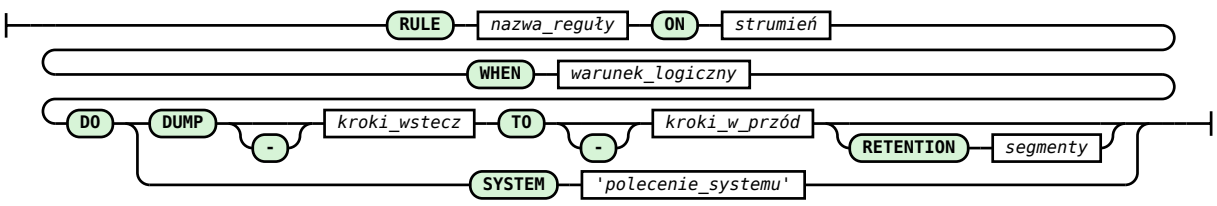
The functionality described here is covered by the tests: `issue42_rule`, described in the appendix Integration Tests.

The syntax of the RULE command is as follows:

```
RULE rule_name
ON data_stream_name
WHEN logical_condition
DO DUMP steps_back TO steps_forward [RETENTION segments]
```

Or like this:

```
RULE rule_name
ON data_stream_name
WHEN logical_condition
DO SYSTEM system_command
```




RULE command syntax diagram

*Fig. 8. RULE command syntax diagram*

The railroad diagram in Fig. 8 was generated from the `rule_statement` rule in the system's ANTLR4 grammar (`RQL.g4`) and covers both forms of the command shown above in a single track: the branch after the word DO leads either to the DUMP variant (with a data-window dump and optional retention), or to the SYSTEM variant (with a system command in quotes). Rounded green boxes are keywords and symbols entered literally, rectangles are values supplied by the user; tracks bypassing the minus sign and the RETENTION clause mean they are optional.

Events defined this way attach to defined data streams. The rule name should be unique. The data stream must be defined before the rule-creation command appears in the rql file.

In both versions of the RULE command, a rule name, a logical condition, and the name of the stream to which the process launched by the DO command is attached are created. The logical condition should refer to variables available in the schema of the data stream following the ON clause.

In the first version of the command, which includes the DO DUMP clause, we define a process that allows collecting data that will arrive in the future. If we omit the RETENTION clause, the dump goes directly to a file named after the rule, prefixed with the stream name. If we add the RETENTION clause, the files will be subject to retention within the range defined by the 'segments' parameter. Sequential numbers will be appended to the end of each dump. Dumps are binary and preserve the schema of all fields of the source data stream. It is worth noting here that the command creates a process in the system which, once the logical condition becomes true, pulls data from the past and also expects its arrival and registration in the future. Nothing prevents us, however, from collecting data only from the past or only from the future. If the step_* values are negative, they refer to the past (i.e. to data that is historical relative to the moment the event described by the logical condition occurred).

The DO SYSTEM clause allows a system event to be triggered once the logical condition based on recorded data is met. This way, an arbitrary system command can be invoked.

Examples of rule declarations in 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"'
```

Assume that a stream str1 has previously been defined, whose data — integer values — arrives once per second. In this case, the first rule, attached to this stream, waits for data whose value exceeds 11. Should such an event occur, a dump of the data is made,

covering the range from 5 seconds before to 5 seconds after the event described by the logical condition.

The second rule, with a slightly different logical condition, prints the text “systemcall” to the screen from which the RetractorDB process was started.

## RULE Command Syntax

The full syntax of the RULE command is:

```
RULE <name>
ON <stream>
WHEN <condition>
DO <action>
```

Where `<action>` can take one of two forms:

```
SYSTEM '<system_command>'
DUMP [-]<step_back> TO [-]<step_forward> [RETENTION <n>]
```

### Restriction

A rule can only be attached to a stream declared with a `SELECT` command (an artifact or substrate). Attaching it to a `DECLARE` input stream is a compilation error:

```
## INVALID — core0 is a declaration, a rule cannot be attached to it
RULE r1 ON core0 WHEN core0[0] > 10 DO SYSTEM 'echo alarm'
```

### The WHEN condition

The condition is a logical expression evaluated to true/false after every new sample of the stream.

Comparison operators: `=`, `!=`, `<`, `>`, `<=`, `>=`. Logical operators: `OR`, `AND`, `NOT`. Examples:

```
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
```

## The DO SYSTEM action

The `DO SYSTEM` action executes the given shell command (via a `system(3)` call) the moment the condition is satisfied. RetractorDB logs the command’s exit code — a non-zero code is reported as an error in the log.

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

Any program available on PATH can be used in the command: shell scripts, Python programs, REST calls, sending notifications, etc.

## The DO DUMP action

The DO DUMP action writes a window of stream samples to a binary file the moment the condition is satisfied. It lets you preserve the context of an event: data before it occurred and data after it.

```
RULE event \
ON results \
WHEN results[0] > 500 \
DO DUMP -10 TO 5
```

Range parameters:

| Parameter | Meaning |
|---|---|
| negative step_back (e.g. -10) | include 10 **historical** samples before the event |
| 0 as step_back | start the dump at the moment of the event |
| positive step_back (e.g. 2) | delay the start of the dump by 2 samples after the event |
| step_forward (e.g. 5) | collect a total of step_forward - step_back samples |

Total number of dumped records: abs(step_forward - step_back). Example: DUMP -5 TO 5 → 10 records (5 historical + 5 subsequent). DUMP 0 TO 1 → 1 record (the current sample).

The step_back range must be less than or equal to step_forward. The step_back value can be negative (history) or non-negative (delay). Both values being negative is not supported.

### Dump files

Files are created in the directory configured by the STORAGE directive. Naming convention:

```
<stream>_<rule_name>_dump.tmp          # without RETENTION
<stream>_<rule_name>_dump_<n>.tmp      # with RETENTION (n = 0..N-1)
```

The file format is raw binary data matching the stream descriptor (no header). The xtrdb tool can be used to read the file.

### The RETENTION option

The RETENTION <n> parameter limits the number of stored dumps — the oldest file is overwritten by the new one (a circular buffer). Without RETENTION, every trigger overwrites a single _dump.tmp file.

```
RULE event \
ON results \
WHEN results[0] > 500 \
DO DUMP -10 TO 5 RETENTION 20
```

The example above keeps the last 20 dumps in files `results_event_dump_0.tmp` … `results_event_dump_19.tmp`.

### Multiple rules for a single stream

Any number of rules of different types can be attached to a single stream:

```
RULE high_alert \
ON measurements \
WHEN measurements[0] > 900 \
DO SYSTEM 'notify-send "Threshold exceeded"'

RULE low_alert \
ON measurements \
WHEN measurements[0] < 10 \
DO SYSTEM 'notify-send "Value too low"'

RULE anomaly_log \
ON measurements \
WHEN measurements[0] > 900 \
DO DUMP -20 TO 10 RETENTION 5
```

All rules for a given stream are evaluated on every new sample.

### Mechanism Construction

By alerting we mean the process of processing current data and having the system react in real time when it recognizes a phenomenon that has occurred. For alerting to work, the system needs mechanisms supporting this process. In RetractorDB I developed an alerting model based on declaring rules tied to the observation of data streams. These rules contain mathematical operations allowing analysis of logical conditions and the triggering of external processes, or performing a data dump within a chosen time window.

**Note**

The functionality described here is covered by the tests: `issue42_rule`, described in the appendix Integration Tests.

The presentation of the RULE command's syntax on page 24 already touches on this functionality. In this chapter I would like to explain in more detail how this solution works.

To build an example demonstrating how alerting works, let's create the following query file — 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"'
```

The file datafile1.txt contains numbers, as text, from 20 to 28.

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

The three commands above declare an ephemeral data source, one data-processing command that shifts it in time by one sample, and an alerting rule. Running the following command:

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

Viewing the resulting out.png file, we will see something like this (Fig. 9):

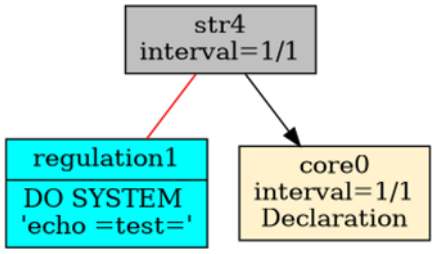



Fig. 9. Dependency between objects when using alerting

The image shows the relationship between the processes responsible for artifacts, alerting, and ephemerides. It should equally be possible to attach the process responsible for alerting to a substrate.

Alerting objects are shown in blue and connected, via red undirected lines, to the objects they monitor.

More than one alerting object can be attached. Multiple RULE commands can be associated with a given data-stream-creating command.

Looking more closely, we see that the process responsible for alerting is triggered by a condition. The following command lets us inspect what's actually happening there:

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

The output file looks as follows (Fig. 10):

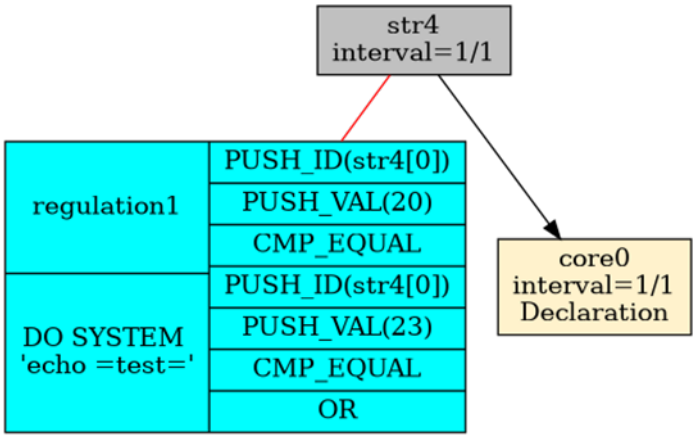



Fig. 10. Code responsible for the alerting trigger condition.

In its final form, this condition must evaluate to an expression representing true or false.

## Logical Condition in RULE

The `WHEN` clause of the `RULE` command takes a logical expression, which is evaluated on every new record of the specified stream. If the expression evaluates to true, the process defined in the `DO` clause is triggered.

### Comparison operators

| Operator | Meaning |
|---|---|
| `=` | equal |
| `!=` | not equal |
| `>` | greater than |
| `<` | less than |
| `>=` | greater than or equal |
| `<=` | less than or equal |

### Logical connectives

| Operator | Meaning |
|---|---|
| `AND` | conjunction — both conditions must be satisfied |
| `OR` | disjunction — one condition is enough |
| `NOT` | negation — the condition must not be satisfied |

### Expression structure

A condition is built from the fields of the stream schema specified in the `ON` clause. Fields are identified the same way as in `SELECT` — by the stream name with an index:

```
WHEN stream[index] operator value
```

Compound conditions are joined with connectives:

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

### Examples

```
RULE high_alarm \
ON measurements \
WHEN measurements[0] > 100 OR measurements[0] < -100 \
DO DUMP -10 TO 10 RETENTION 50

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

RULE one_time \
ON data \
WHEN NOT data[0] = 0 \
DO DUMP -5 TO 0
```

### Field access

The condition refers to the fields of the stream specified in `ON`. The field index corresponds to its position in that stream's schema — the same as in the `SELECT` clause. Aliasing works exactly as described in the chapter Aliasing.

## Alerting Example

In a terminal window, we start the xretractor process, running the query.rql file shown at the start of the chapter.

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

In a second terminal window, I suggest running the command:

```
$ xqry -s str4
27
28
20
21
```

```
22
23
24
25
26
27
```

I suggest placing both windows side by side. We will see that the appearance of the values 20 and 23 triggers the server-side action that prints “test”. Keep in mind that any system command or invocation of any program can appear here, depending on what we put in the DO SYSTEM declaration.

### Example 2: recording event context (DO DUMP)

The `DO DUMP` action lets you capture a window of samples surrounding an event — data before and after it occurred. This is useful when we want to preserve the context of an anomaly for later analysis.

We create a `query.rql` file:

```
STORAGE 'temp'

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

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

Input data — numbers from 20 to 28:

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

We run xretractor:

```
$ xretractor query.rql
```

When the value of stream `str1` exceeds 24, the rule triggers a write of 6 records (3 historical + 3 subsequent) to the binary file `temp/str1_anomaly_log_dump.tmp`.

**Reading the dump file** The dump file contains no `.desc` header — when opening it in `xtrdb`, the schema must be specified manually:

```
$ xtrdb
> storage temp
> open str1_anomaly_log_dump { INTEGER a }
> size
> list 6
> quit
```

### Example 3: rotating dumps (DO DUMP with RETENTION)

Without `RETENTION`, each successive trigger of the rule overwrites the same file. When events repeat, use `RETENTION N` to keep the last N dumps in separate files.

```
STORAGE 'temp'

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

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

Each trigger creates the next file (circular rotation):

```
temp/str1_anomaly_log_dump_0.tmp
temp/str1_anomaly_log_dump_1.tmp
temp/str1_anomaly_log_dump_2.tmp
temp/str1_anomaly_log_dump_3.tmp
temp/str1_anomaly_log_dump_4.tmp
```

Once capacity is exceeded (`RETENTION 5`), the oldest file is overwritten by the new one.

### Example 4: multiple rules on a single stream

Any number of rules can be attached to a single stream. The example below combines both actions — a system notification and context recording:

```
STORAGE 'temp'

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

RULE lower_threshold \
ON str1 \
WHEN str1[0] < 21 \
DO SYSTEM 'echo "ALARM: value below lower threshold" >> alarm.log'

RULE upper_threshold \
ON str1 \
WHEN str1[0] > 26 \
DO SYSTEM 'echo "ALARM: value above upper threshold" >> alarm.log'

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

The `upper_threshold` and `context_log` rules react to the same condition independently — crossing the upper threshold both writes a log entry and captures the data window at the same time. The `lower_threshold` rule handles the lower threshold separately.

All three rules are evaluated on every new sample of the `str1` stream.

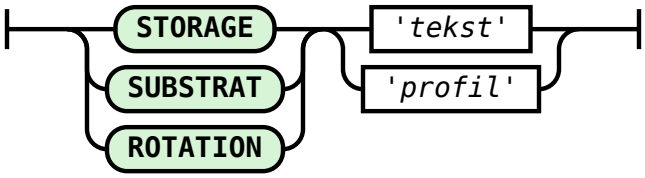
## 3.4. Configuration Directives

At present, three configuration directives have been developed.

- STORAGE
- SUBSTRAT
- ROTATION

Each of these directives is followed by a text string enclosed in double or single quotes. An example use of both configuration directives is as follows:

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




Configuration directive syntax diagram

*Fig. 11. Configuration directive syntax diagram*

The railroad diagram in Fig. 11 was generated from the `compiler_option` rule in the system's ANTLR4 grammar (`RQL.g4`). All three directives have an identical structure: one of the keywords STORAGE, SUBSTRAT, or ROTATION (rounded green boxes), followed by a value enclosed in single quotes — arbitrary text (a directory path for STORAGE, a counter-file name for ROTATION), or the name of one of the predefined memory profiles (for SUBSTRAT).

Storage is used to indicate the system directory in which all output files should be created. Without this directive, files created by the system are placed, by default, in the current directory from which the main RetractorDB process was started.

Substrates are queries and their effects that arise from the compiler decomposing system commands based on time-series algebra expressions. These are queries visible in the query execution plan but not specified directly in the .rql file. They arise from the implementation of the query-execution-plan construction process.

By default, such queries materialize data on disk in the form of unbounded files. This behavior can be desirable during software development, but once the system is deployed in a production environment it is better to keep substrates in temporary memory areas.

The possible options for the SUBSTRAT command are: memory, default, direct, posix, posixshd, generic, device, textsource. A full description of each type — the C++ class, retention handling, and shadow support — can be found in the chapter Storage Types.

The last directive — Rotation — indicates an alternative shutdown mode for the system. By default, after compilation, all files produced by the system remain in whatever state the system left the recorded data. On the next invocation of the system command, all artifact and substrate files are deleted. Using the Rotation directive in an rql file containing query declarations makes the system create the file named in the directive's parameter and store there a counter incremented on every system startup. Artifact and substrate files are renamed at the end of every system run — they get an `.old` extension plus a number derived from the increasing counter. This process is called artifact rotation.

# Chapter 4

# System Architecture

The construction of the data-processing system is a strictly technical chapter. Here I present how the system was designed and built, and where and how its functionality is currently laid out.

RetractorDB was implemented in C++ under Linux. The source code goes through continuous integration and testing on GitHub, backed by CircleCI. The code is run and developed locally on the Linux WSL2 platform. I abandoned development and implementation of the system under Windows. In the early phase I kept that option open, and I may return to it in the future. However, maintaining too many development platforms significantly slows down rapid prototyping and system development. I still keep and maintain the system's functionality on the Linux ARM platform. The code is compiled and tested on ARM and x86-64 architecture machines running in CircleCI's resources. Raspberry Pi is one of the intended production platforms for RetractorDB, aimed at Edge IoT needs.

Compilation of the system's code is supported by the Conan package manager [8]. If we want to understand how the toolchain that builds the system's code is put together, we can look at the file /.circleci/config.yml, which contains the procedure for building and running the system in CircleCI's container/machine environment. The files `/docker/ci/Dockerfile` and `/docker/ci/DockerConan.txt` contain instructions on how the container image that builds the system with preconfigured dependencies is built. Reviewing these files will show what is needed, and how to install it on your own system, in order to compile the system's sources locally.

## 4.1. Overview of topics covered in this chapter

The chapter is built in layers — from the general view down to implementation detail.

### General Perspective

The system as a trio of cooperating programs: `xretractor` as the singleton executing the query plan, `xqry` as the multi-instance client for current data, `xtrdb` as the binary-file inspection tool. Communication between the `xretractor` and `xqry` processes is

carried out through shared memory (Boost IPC). The diagram in Fig. 12 shows the boundary of responsibility for each component.

### Data and Control Flow

Which data paths are always active (data arrival → xretractor → artifacts), and which are optional or diagnostic. The graceful-shutdown mechanism is also described — xretractor reacts to `SIGINT`, `SIGTERM`, and `SIGHUP` signals by finishing the current cycle without risking file corruption.

### Artifacts, Substrates, and Ephemerides

The system's key taxonomic split. Each stream type has a different purpose and a different storage strategy: artifacts are materialized on disk as a durable result, substrates are intermediate streams necessary during computation, and ephemerides are ephemeral data sources that cannot, or need not, be stored.

### Data Storage Format

The four-file structure of an artifact: a binary data file (fixed-length records, no header), a `.desc` descriptor describing the record schema in ANTLR4 grammar, a `.meta` metadata file with an index of null values and transmission gaps (RLE encoding), and an optional `.shadow` file for non-destructive modification of historical records. The descriptor determines the storage strategy via the `TYPE` field.

### Compilation and Plan Construction

The process of turning an `.rql` file into a ready-to-run query execution plan. The `-c` flag runs compile-only mode without execution; combined with `-d -f -s` it generates DOT output, which `graphviz` turns into a data-flow graph. The graph shows two domains: the arithmetic-expression stack (PUSH, ADD, etc.) and the stream algebra. The full set of compile-mode and execution-mode flags is described.

### Data Processing and Distribution

A complete walkthrough: from preparing a data file, through running `xretractor`, through viewing streaming statistics (`xqry -d`), to live visualization in gnuplot (`xqry -s str1 -p 50,50 | gnuplot`) and network transmission via `nc`. The example combines two sources — a text file and `/dev/urandom` — illustrating how the `+` operator in the FROM clause performs algebraic stream joining.

### Artifact Analysis

The `xtrdb` tool — an interactive binary-file inspector modeled after the dbase style. The `.open`, `.desc`, `.list`, `.rlist`, and `.meta` commands let you browse the contents of artifacts without knowing the binary format. The tool is also used to verify determinism: the same input data should always produce identical results.

Three commands are enough to run a complete pipeline:

```
xretractor -c query.rql          # verify the query file's correctness
xretractor query.rql             # start processing
xqry -s <stream>                 # read current data
```

A fourth element — `xtrdb` — comes into play for diagnostics and testing, not in a typical production workflow.

## 4.2. General Perspective

The system is built around 3 programs available as system commands. The first is the compiler and query-plan execution engine. The second is the client for accessing current data. The third is the program that provides access to binary dumps. Their names are, in order:

- xretractor
- xqry
- xtrdb

The xretractor program creates RetractorDB's main process. The xqry program creates processes that communicate with RetractorDB. Communication happens through a shared memory area. The xtrdb program is used to analyze data and metadata stored in the database's files.

Below, Fig. 12 schematically shows RetractorDB's architecture. All currently existing components are included. The areas enclosed in boxes with headers filled with system commands correspond to the existing components. The artifact-storage area is a symbolic representation of the filesystem.

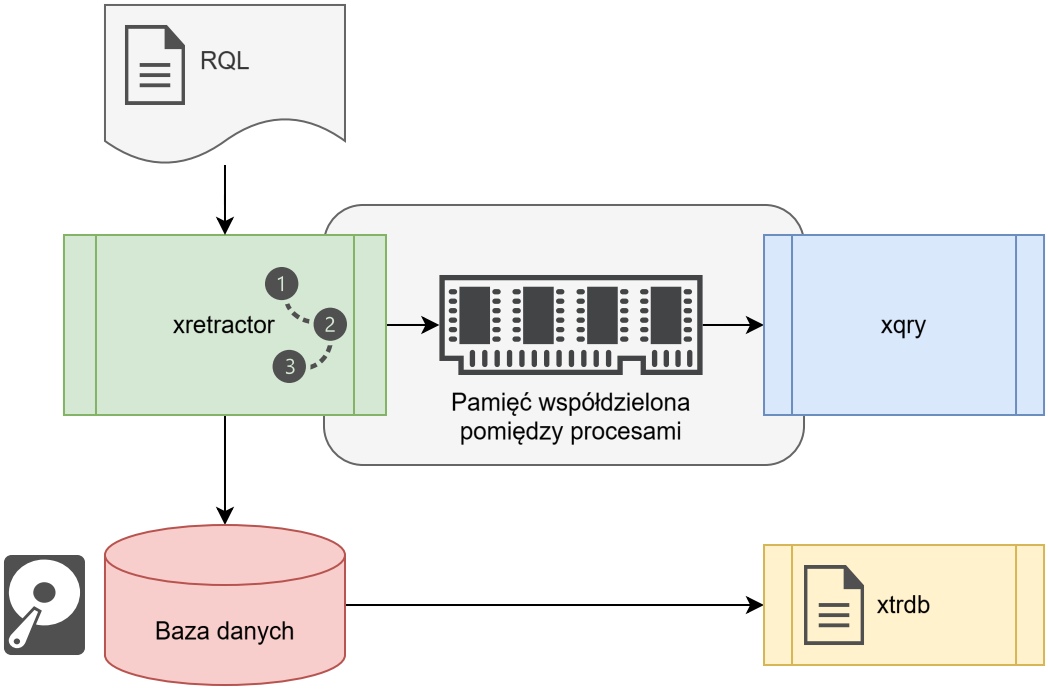



Fig. 12. Data-flow diagram between RetractorDB processes

In Fig. 12 we see the processes carried out by the xretractor, xtrdb, and xqry programs. The figure schematically shows how the processes in RetractorDB communicate. It shows the shared parts of the developed tools.

The xretractor process communicates with xqry processes through a shared memory area. In this memory, xretractor creates a data queue for every xqry process. Data is received by xqry processes on an ongoing basis. The job of the xqry processes is to forward the data on to other systems or processes. If an xqry process dies or is terminated, xretractor, which manages the shared area, frees the area dedicated to it within the shared region.

Besides directing data for delivery through shared memory, RetractorDB also writes data to the so-called artifact-storage area. Currently this is a directory to which the results of the stream-processing carried out according to RetractorDB's query execution plans are continuously written.

**Warning**

The "Database" shown in the figure is not a relational database. By "database" in the figure shown, we mean a set of binary or text files managed by RetractorDB. Data is pulled from devices and written to rotating or non-rotating binary or text files. Access to this data is carried out via the xtrdb tool, or, while the system is running, via the xqry process.

The file with RQL queries and directives is given as the required first argument to the command that starts the system. This behavior will probably change in the future — eventually the system should start as a service and wait for the operator to supply

a file with directives. For now, though, we start the system with an initial input. If you want to add something while the system is running, see the chapter titled Ad Hoc Queries.

## 4.3. Data and Control Flow

Data and control in RetractorDB give rise to several potential ways of using the system's components. Fig. 13 schematically shows the flow of data between RetractorDB's processes, Linux system processes, and the source data and results produced by each process.

The thickest lines represent the flow that is always present in the process of processing regular time series. To start, the xretractor process currently needs an .rql file containing a sequence of queries. After compiling, the xretractor process builds the query plan tree and begins processing incoming data and creating binary files containing artifacts.

**Note**

The functionality described here is covered by the test: `consistency`, described in the appendix Integration Tests.

To control the xretractor process once it has started, we use the xqry process. Through it, we can stop the xretractor process, retrieve statistics, or request access to current data.

The remaining arrows represent data flows that depend on the specific process being carried out with RetractorDB. Dashed arrows are typically intended for diagnostic purposes.

Each process on the diagram is additionally labeled with the number of continuous processes of that kind maintained in the system. The label "1" next to the xretractor process means that this program ensures only one instance of this process runs in the system. Attempting to start another one will fail with an error message at startup. The xtrdb program does not maintain any continuous or unbounded processes. It reads data, processes it, returns results, and exits. It also offers an interactive mode. The xqry process is labeled "N" — meaning that more than one xqry process can be invoked. This is a typical usage scenario for RetractorDB. By definition, there can be several clients communicating with the query-execution-plan processor.

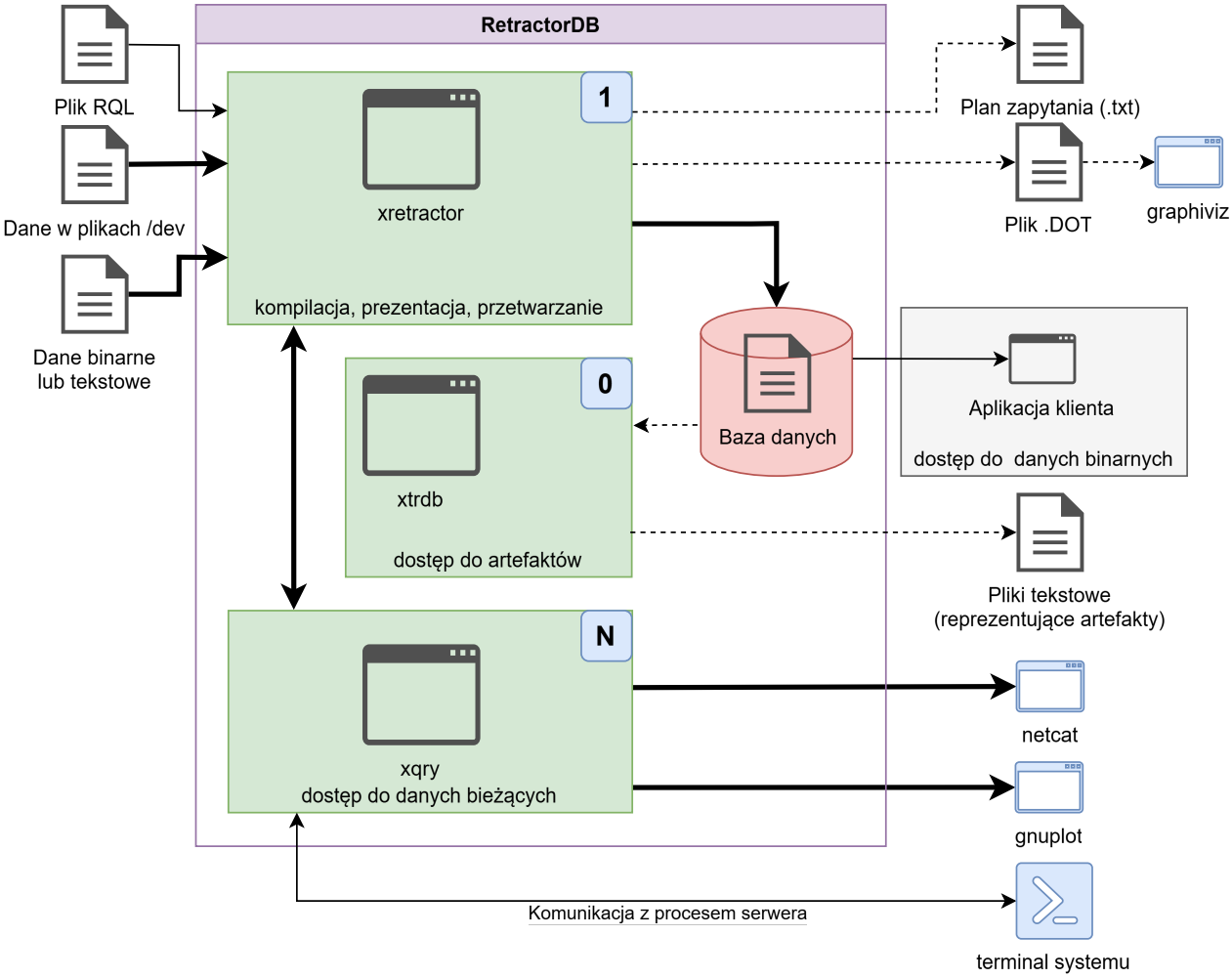



Fig. 13. Data and control flow

## Stopping xretractor

The xretractor process handles system signals and shuts down in a controlled manner upon receiving:

| Signal | Command | Meaning |
|---|---|---|
| `SIGINT` | Ctrl+C in a terminal | interactive interrupt |
| `SIGTERM` | `kill <pid>` | standard process termination |
| `SIGHUP` | `kill -HUP <pid>` | termination on terminal close |

All three signals produce the same effect: a graceful shutdown — the processing loop finishes the current cycle and stops. This allows xretractor, running as a service, to be shut down safely without risking corruption of artifact files.

### Stopping via xqry

Besides system signals, xretractor can be stopped programmatically — using the command:

```
xqry --kill
```

**How the shutdown proceeds step by step** **1. xqry sends a "kill" request**

The xqry process builds an IPC message and places it on the `RetractorQueryQueue` message queue — the shared channel connecting all clients to xretractor. The message contains the xqry process's identifier (PID) and the `kill` command.

**2. xretractor receives the command and sets the stop flag**

xretractor's communication thread (`commandProcessorLoop`) continuously listens on `RetractorQueryQueue`. Upon receiving a `kill` message, it sets the atomic variable `iTimeLimitCnt` to `stop_now`. The same mechanism is used by the system-signal handler — regardless of the source (a `SIGINT`/`SIGTERM`/`SIGHUP` signal, or the xqry `--kill` command), the effect is identical.

**3. The main processing loop detects the flag and finishes the current cycle**

The main loop checks `iTimeLimitCnt` on every iteration. When it detects the value `stop_now`, it finishes the current cycle and exits the loop — without interrupting mid-computation. This ensures the integrity of the artifacts being written.

**4. xretractor notifies all connected clients (OOB broadcast)**

After exiting the loop, xretractor calls `boradcastOutOfBussiness()`. This function walks the internal `id2StreamName_Relation` map, which contains an entry for every xqry process subscribed to a data stream (every `xqry --select` invocation registers itself in this map via the `show` command). For every registered client, xretractor sends a special `OUT_OF_BUSSINESS` message to its dedicated queue.

**5. Every xqry client receives the termination signal and exits**

Every running xqry process has its own individual message queue named `brcdbr<PID>`. Upon receiving the `OUT_OF_BUSSINESS` message, xqry sets its internal `done` flag and shuts down in a controlled manner — regardless of how much data it had received up to that point.

**6. IPC resource cleanup**

Finally, xretractor removes all shared IPC resources: the `RetractorShmemMap` shared-memory segment, the `RetractorQueryQueue` command queue, the `RetractorMapMutex` mutex, and every client's individual queue.

**What happens with multiple xqry processes** RetractorDB is designed to work with multiple parallel clients. If, say, three xqry processes are running simultaneously, subscribed to different streams, and one of them calls `xqry --kill`:

- xretractor processes the kill request **once**, regardless of which client sent it,

- the `boradcastOutOfBussiness()` mechanism sends the `OUT_OF_BUSSINESS` message to **all** registered clients at the same time,
- each of the three xqry processes receives the termination signal and exits on its own,
- clients that hadn't subscribed to any stream (e.g. xqry invoked only with `--dir` or `--hello`) are not entered in the map and don't need to be notified — these commands exit immediately after providing their response.

It's worth noting that xqry also detects server inactivity: if no data arrives for 10 seconds, the client shuts itself down with a warning in the log. This is a safeguard in case xretractor crashes suddenly without being able to send the OOB message.

## 4.4. Artifacts, Substrates, Ephemerides

Given that the system is designed for continuous operation, and that, theoretically, the results obtained without a data-retention process would fill up any storage medium, we introduce additional definitions related to the nature of the data being processed.

In presenting the description of Fig. 13, artifacts were mentioned. This is one of the terms that needs explaining.

**Note**

Definition (Artifact): By artifacts we mean data processed in the system in the form of streams that are ultimately materialized as a durable result and effect of processing other data.

Continuous time series can be read from devices, then processed — reduced or resized in time and in dimension. But as a rule, certain data should be saved. Whether that data is later subject to retention is a secondary matter. Data that constitutes the result and expected output of the system, we call artifacts. Something we expect and materialize for the end user's needs.

**Note**

Definition (Substrate): Substrates are intermediate objects. As a result of processing time series, data streams may arise that are ephemeral — needed only for and during processing.

Their size can be significant, considering how far back we go, for example when dumping historical monitoring data. However, their existence is irrelevant with respect to the system's desired output. We call such data streams substrates. They arise as a result of the system's operation; as a rule they do not appear explicitly in queries — but they result from the process of processing time series, and yet their results are necessary to accomplish the task.

**Note**

Definition (Ephemeris): Ephemerides are objects on the basis of which we create source data streams — data that cannot be stored. As a rule, these are ephemeral, transient data.

For example, the system reads random numbers at a given rate, and it is precisely this data source that provides ephemeral data. It cannot be returned; storing it is, as a rule, pointless — it must be passed on for further processing in order to produce artifacts or substrates, and then discarded and replaced with new, current data.

## 4.5. Data Storage Format

The system processes time series in three forms: **artifacts**, **ephemerides**, and **substrates**. Each type has a different purpose and a different storage strategy.

Substrates and Artifacts are formally no different in the system. The only difference is that substrates were generated based on data-stream algebra equations and were not written directly in the sequence of commands given to the compiler. If we declare an Artifact stream that covers what would otherwise be a substrate, the substrate is eliminated. Ephemerides are streams created via the Declare command — they contain values that exist only briefly.

### Storage accessor types

**Note**

The functionality described here is covered by the test: `txtsrc`, described in the appendix Integration Tests.

The `TYPE` field in the descriptor (or the `STORAGE` directive in RQL) selects the `FileInterface` implementation:

| Type (`TYPE_PROFILE`) | Implementation class | Purpose |
|---|---|---|
| `DEFAULT` | `groupFile<posixBinaryFileWithShadow>` | Default artifacts — data file + shadow file, with retention |
| `DIRECT` | `groupFile<posixBinaryFile>` | Direct writes without shadow, with retention |
| `POSIX` | `posixBinaryFile` | Raw POSIX write, no shadow |
| `POSIXSHD` | `posixBinaryFileWithShadow` | POSIX with a shadow file |
| `MEMORY` | `memoryFile` | RAM-only storage (ephemerides) |
| `GENERIC` | `genericBinaryFile` | Generic binary accessor |
| `DEVICE` | `binaryDeviceRO` | External binary input-data device (read-only) |

| Type (TYPE_PROFILE) | Implementation class | Purpose |
|---|---|---|
| TEXTSOURCE | textSourceRO | Text input-data source (read-only) |

---

### The artifact and substrate file set

Artifacts and substrates written to disk can be associated with up to four files:

| File | Extension | Purpose |
|---|---|---|
| Binary data file | *(stream name)* | The main record stream — append-only |
| Descriptor file | .desc | Record schema (fields, types, sizes, storage type) |
| Metadata file | .meta | Index of null values and transmission gaps (RLE) |
| Shadow file | .shadow | Record modifications without overwriting original data |

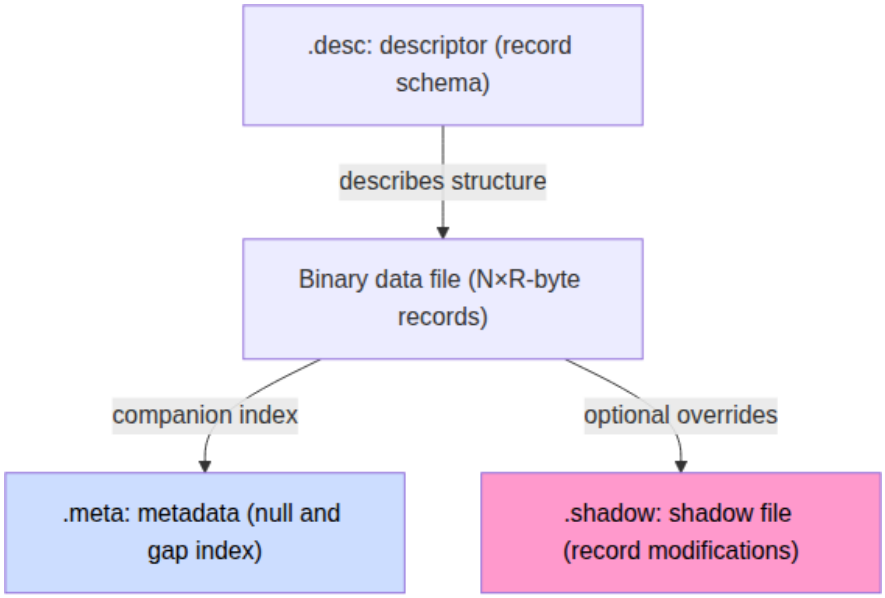



Fig. 14. The artifact file set and their relationships

The diagram in Fig. 14 shows the static relationship between artifact files: .desc defines the record structure, .meta indexes nulls and gaps, and .shadow stores optional record overrides.

The shadow file and the metadata file are optional. With continuous, gap-free, unmodified data arrival, the binary data file and the descriptor alone are enough.

Ephemerides **have no files on disk at all** — they exist only in the process's working memory and disappear once it ends.

---

## Chapters

- Artifact Files — descriptor, binary data, metadata, shadow file, and the relationships between them
- File Rotation Mechanism — the `ROTATION` directive, file lifecycle, session examples
- Inspection Tool `xtrdb -s` — the storage map, report sections, examples
- Summary — rationale for the chosen structure, comparison of approaches

## Files

This chapter describes the five files that make up the complete file set of an artifact or substrate: the schema descriptor (`.desc`), the main binary data file, the metadata index (`.meta`), the data shadow file (`.shadow`), and the index shadow file (`.meta.shadow`). For each file, the binary format, field semantics, and read/write rules are presented. The chapter also covers the `metaDataStream` class — the RLE compression mechanism, transmission-gap handling, the update interface, and persistence across restarts. The final section shows the relationships between all the files at the level of `append`, `update`, and `read` operations.

The scope of this chapter **does not include** the file-rotation mechanism between sessions (→ Rotation) or the `xtrdb -s` inspection tool (→ Inspection Tool).

---

### The descriptor file (.desc)

The `.desc` file describes the record structure. It is parsed by an ANTLR4 grammar (`DESC.g4`) and can contain data fields, storage-type meta-information, and a retention policy.

#### Syntax

```
{ <statement>* }
```

Each statement is one of the following:

```
BYTE     name [N]          # array of N bytes (default N=1)
INTEGER  name [N]          # 32-bit signed integers
UINT     name [N]          # 32-bit unsigned
FLOAT    name [N]          # 32-bit floating point (IEEE 754)
DOUBLE   name [N]          # 64-bit floating point
RATIONAL name [N]          # pair of int64: numerator and denominator
STRING   name [size]       # fixed-length string
REF      "path/file"       # reference to an external descriptor file
```

```
TYPE     identifier        # storage type (DEFAULT, MEMORY, POSIXSHD, …)
RETENTION capacity segment # cyclic on-disk retention
RETMEMORY capacity         # cyclic in-memory retention
```

**Example `.desc` files** **A default artifact** — two numeric fields, `DEFAULT` storage (data file + shadow file):

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

**An ephemeris** — an ephemeral, RAM-only stream:

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

**A substrate with retention** — a cyclic on-disk buffer of the last 1000 records (10 segments of 100):

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

**A binary source declaration** (`DECLARE` in RQL generates this schema):

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

**Field type sizes**

| Type | Size of a single value |
|---|---|
| `BYTE` | 1 B |
| `INTEGER` | 4 B |
| `UINT` | 4 B |
| `FLOAT` | 4 B |
| `DOUBLE` | 8 B |

| Type | Size of a single value |
|---|---|
| `RATIONAL` | 16 B (two int64) |
| `STRING` | N B (declared size) |

For array fields `name[N]`, the total size = type_size × N. The `TYPE`, `REF`, `RETENTION`, and `RETMEMORY` fields take no space in the record — they are descriptor metadata.

Record size `R` = the sum of the sizes of all data fields.

**The TYPE field and storage strategy** The `TYPE` field in the descriptor directly determines which accessor (`FileInterface`) is used by `storage::initializeAccessor()`. The absence of a `TYPE` field is equivalent to `DEFAULT`. The value is case-insensitive (`MEMORY` = `memory`).

---

### The binary data file

The data file is a sequence of fixed-length records, written one after another with no header at all. The size of a single record `R` is determined by the descriptor as the sum of all field sizes in bytes.

| Offset in file | Content | Size |
|---|---|---|
| 0 | Record 0 | R bytes |
| R | Record 1 | R bytes |
| 2R | Record 2 | R bytes |
| … | … | … |
| (N-1) × R | Record N-1 | R bytes |

Every record contains the packed field values in the order defined by the descriptor:

| Offset in record | Field | Size |
|---|---|---|
| 0 | field_0 | len_0 bytes |
| len_0 | field_1 | len_1 bytes |
| len_0 + len_1 | … | … |
| len_0 + len_1 + … + len_n | field_n | len_n bytes |

The **append** operation (adding a new record) writes data to the end of the file. The **update** operation (modifying an existing record) — if a shadow file exists — goes into the shadow file, not the main file.

**Example**

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

Record size: INTEGER (4 B) + FLOAT (4 B) = **8 bytes**. After 5 seconds of data arriving (10 Hz), the file `data.dat` is 5 × 10 × 8 = **400 bytes**.

---

### The metadata file (.meta)

The `.meta` file is an index of null values and transmission gaps. It stores information about which record fields have null values, and where gaps occurred — without duplicating the data itself.

**File format**

| Position | Content | Size |
|---|---|---|
| Header | `creationTimeNs` (int64) | 8 bytes |
| RLE entry 0 | `gapFlag \| count \| bitsetSize \| bitset` | variable |
| RLE entry 1 | `gapFlag \| count \| bitsetSize \| bitset` | variable |
| … | … | … |
| RLE entry k | the current in-memory entry | variable |

**RLE entry format** Each entry describes a run of consecutive records with an identical null pattern:

| Field | Size | Description |
|---|---|---|
| `gapFlag` | 1 B | 0 = normal record, 1 = gap |
| `recordCount` | 8 B (size_t) | number of records in the run |
| `bitsetSize` | 8 B (size_t) | number of fields (N) |
| `bitset` | ⌈N/8⌉ B | bit i = field i is null |

**RLE compression** Consecutive records with the same null pattern are merged into a single entry by incrementing `recordCount`. A new entry is only created once the pattern changes.

**10 records, 2 fields, no nulls:**

| Entry | isGap | count | bitset |
|---|---|---|---|
| entry 0 | F | 10 | [F,F] |

**Field 1 becomes null starting at record 5:**

| Entry | isGap | count | bitset |
|---|---|---|---|
| entry 0 | F | 5 | [F,F] |
| entry 1 | F | 5 | [F,T] |

**Transmission gap after record 3:**

| Entry | isGap | count | bitset |
|---|---|---|---|
| entry 0 | F | 3 | [F,F] |
| entry 1 | T | 7 | [T,T] |
| entry 2 | F | … | [F,F] |

**The transmission-gap marker** A transmission gap (e.g. a system shutdown, a lost signal) is recorded as an entry with `isGap=true` and all null bits set to `true`. The `count` parameter stores the length of the gap in units of the stream's interval. The binary data file itself contains no additional records for the gap — that information lives solely in the `.meta` file.

**Note**

The functionality described here is covered by the tests: `issue113_meta_internal`, `issue113_meta_autocreate`, described in the appendix Integration Tests.

---

**The metaDataStream class** The `.meta` file is managed by the `rdb::metaDataStream` class. It encapsulates three areas of responsibility:

1. **In-memory RLE aggregation** — it buffers the current segment (the most recent run of records with an identical null pattern) in the `currentEntry_` field, without writing it to the file on every record.
2. **Data persistence** — only completed segments (when the pattern changes, or on an explicit call to `flushCurrentEntry()`) are written to the file as committed entries.
3. **A query index** — it exposes an interface for querying the null pattern of any record and for detecting transmission gaps.

The class holds two states:

| State | Location | Description |
|---|---|---|
| Committed segments | the `.meta` file on disk | all completed RLE runs |

| State | Location | Description |
|---|---|---|
| Current segment (`currentEntry_`) | working memory | the run currently being accumulated (not yet written, or subject to being overwritten) |

**Object lifecycle** The state diagram (Fig. 15) shows the transitions between phases of a `metaDataStream` object:

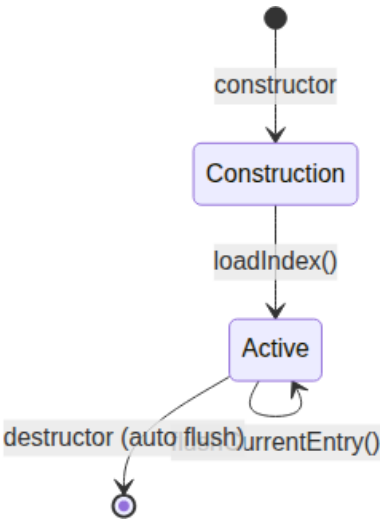



Fig. 15. Lifecycle of a metaDataStream object

**Constructor** (`metaDataStream(descriptor, path)`): - Initializes an empty `currentEntry_` based on the number of fields in the descriptor. - Calls `loadIndex()` — if the file exists, it loads all committed segments, determines `committedRecordCount_`, and moves the last non-gap segment back into `currentEntry_` (allowing the RLE run to continue after a restart). - If the file does not exist, it creates it and writes the header (the stream's creation timestamp).

**The destructor** automatically calls `flushCurrentEntry()`, guaranteeing that the current buffer reaches disk even when the program exits normally.

**The update interface** The class distinguishes three scenarios for changing metadata state:

**`onRecordAppended(nullBitset)`** Called by `storage` after every new record is appended to the data file.

```
pattern identical to currentEntry_?
├─ YES → increment currentEntry_.recordCount (RLE accumulation, no I/O)
└─ NO  → flushCurrentEntry() (previous segment to disk)
          set currentEntry_ = {nullBitset, count=1}
```

I/O only happens **when the pattern changes** — for a run of identical records, the cost is a single in-memory counter increment.

**`onRecordModified(index, nullBitset)`** Called by `storage` when updating an existing record. Behavior depends on the operating mode:

**Normal mode** (no data shadow file): it locates the record within the RLE segments and splits the segment into up to three parts: before the modified record, the record itself, and after it.

```
is the record in currentEntry_ (memory)?
├─ YES → splitSegment() in memory, new fragments appended to the file
└─ NO  → read the file, splitSegment(), rewrite the file (rewriteFile)
```

Example of splitting a segment `[allNull × 5]` when modifying record 2:

```
Before: [allNull × 5]
After:  [allNull × 2] [allPresent × 1] [allNull × 2]
```

**Shadow mode** (`shadowMode_ = true`, activated via `setShadowMode(true)`): instead of modifying the main index, it appends a single null-pattern override to the `.meta.shadow` file. The main `.meta` index remains untouched and consistent with the main data file.

```
shadowMode_?
├─ YES → appendShadowOverride(index, nullBitset) → entry in .meta.shadow
└─ NO  → applyModificationToMainIndex(index, nullBitset) → splitSegment()
```

**`onTransmissionGap(duration)`** Records a transmission gap of the given length (in units of the stream's interval). It first commits the current segment (`flushCurrentEntry()`), then appends an entry with `isGap=true` to the file (Fig. 16).

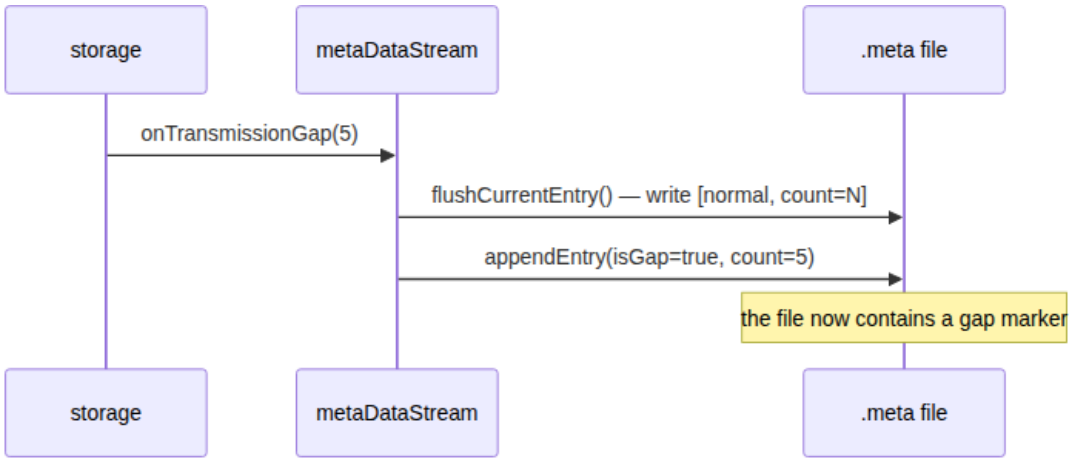



Fig. 16. Gap-recording sequence — onTransmissionGap

**Safety mechanism: `flushCurrentEntry()` and overwriting (tailDirty_)** The `storage` class calls `flushCurrentEntry()` after **every** call to `write()`, to guarantee survival of a process crash. A naive implementation would append a new entry to the file on every flush — causing file growth proportional to the number of records, even without any change in the null pattern.

The solution: a **lazy overwrite** mechanism flagged by `tailDirty_`.

```
flushCurrentEntry() → write [pattern, count=2] to disk
onRecordAppended(the same pattern):
    currentEntry_.count = 2 (restored from disk)
    tailDirty_ = true        ← the next flush will overwrite, not append
    currentEntry_.count++    → count = 3
flushCurrentEntry() → seek to the last entry, overwrite [pattern, count=3]
    (file size unchanged)
```

The sequence diagram for `storage`'s typical pattern (append + flush after every record) is shown in Fig. 17:


Fig. 17. The lazy-overwrite mechanism — overwriting the last .meta entry

Thanks to this, the `.meta` file grows only when the **null pattern changes** — not on every record. With continuous, uniform data arrival, the file has a constant size regardless of the number of records.

**Persistence and state recovery** After the process restarts, a new `metaDataStream` object loads the file via `loadIndex()` (sequence shown in Fig. 18):

1. It reads the header — the timestamp (`creationTimeNs`), stored as `int64` nanoseconds since the epoch.
2. It loads all committed entries from the file.

3. If the last entry **is not a gap**, it moves it back into `currentEntry_` and removes it from the file (allowing the RLE run to continue after a restart without duplication).
4. It computes `committedRecordCount_` as the sum of `recordCount` over all non-gap entries remaining in the file.

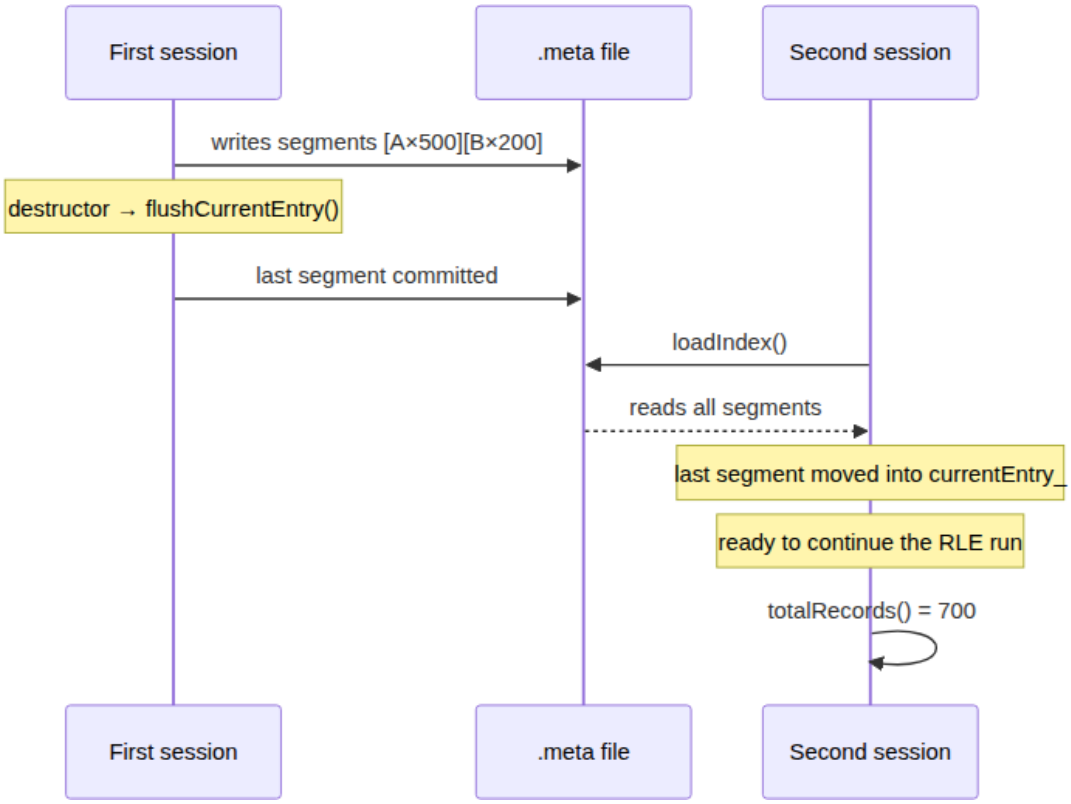



Fig. 18. Persistence and state recovery after a restart

### Query interface

| Method | Description |
|---|---|
| `getNullBitset(i)` | Returns the null pattern for record `i`. In shadow mode, it first checks overrides in `shadowOverrides_` (from the end — the most recent wins), and only falls back to the main index if there's no entry. |
| `isGapBefore(i)` | Returns `true` if, in the RLE index, an entry with `isGap=true` sits immediately before record `i`. Record 0 never has a gap before it. |
| `segments()` | Returns all RLE segments: committed (from disk) plus the current one (from memory), if non-empty. Does not include overrides from `.meta.shadow`. Used for inspection and tests. |
| `totalRecords()` | The sum of records across all segments (committed + pending). |
| `isEmpty()` | Shorthand for `totalRecords() == 0`. |

| Method | Description |
|---|---|
| `rotate(percounter)` | Rotates the index file: renames the current `.meta` file to `.meta.old<N>`, creates a new empty file. Called by `storage::detectStartupState()` after detecting data-file rotation (data file empty, index non-empty). When `percounter < 0`, the file is not renamed — only an index reset is performed. |
| `reset()` | Clears the index in place: zeroes the counters, rewrites the file with only the header, without renaming it. Also calls `discardShadow()`. Called by `storage` when clearing without preserving history (e.g. after `purge()`). |

**The index-shadow interface** A set of methods for managing the `.meta.shadow` file. Called by `storage::attachStorage()` and the related operations on the data shadow file.

| Method | Description |
|---|---|
| `setShadowMode(enabled)` | Enables or disables shadow mode. With `enabled=true` it calls `loadShadow()` — loading existing overrides from the `.meta.shadow` file. |
| `mergeShadow()` | Merges the shadow overrides into the main index (calling `applyModificationToMainIndex()` for each override, in write order — the last one wins), then deletes the `.meta.shadow` file. The counterpart to `merge()` for the data shadow file. |
| `discardShadow()` | Clears the in-memory list of overrides and deletes the `.meta.shadow` file. Called when discarding the data shadow (purge, reset, rotation). |

**Usage example — a typical production scenario**

```
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()

The .meta file after the above operations (4 flushes, 2 segments):
  [isGap=F, count=2, bitset=[F,F,F]]   ← entry 0
  [isGap=F, count=1, bitset=[T,F,F]]   ← entry 1  (rec2)
 [isGap=F, count=1, bitset=[F,F,F]]  ← entry 2  (rec3, currently in memory)

getNullBitset(2) → [T,F,F]   (field 0 of record 2 is null)
isGapBefore(2)  → false
totalRecords()  → 4
```

### The shadow file (.shadow)

The shadow file allows modification of recorded records without destroying the original data. Deleting the `.shadow` file restores the original state of the data.

### Entry format

| Field | Size | Description |
|---|---|---|
| `position` | 8 B (size_t) | the record's index in the main file |
| `data` | R bytes | the record's new values |

Every modification appends a new entry to the end of the shadow file. With multiple modifications of the same record, the file may contain multiple entries for the same position — the most recent one is the current one.

**Read priority** Read priority is the rule for deciding which source the system should return a record's value from, when the same index could appear in both the main file and the shadow file at once. In RetractorDB, priority is defined deterministically: `.shadow` is checked first (from the end, to pick the most recent modification), and only if there's no entry is a read performed from the main file. This concept concerns the **consistency and read versioning** of data after modifications, not the physical record-storage format of the binary file itself.

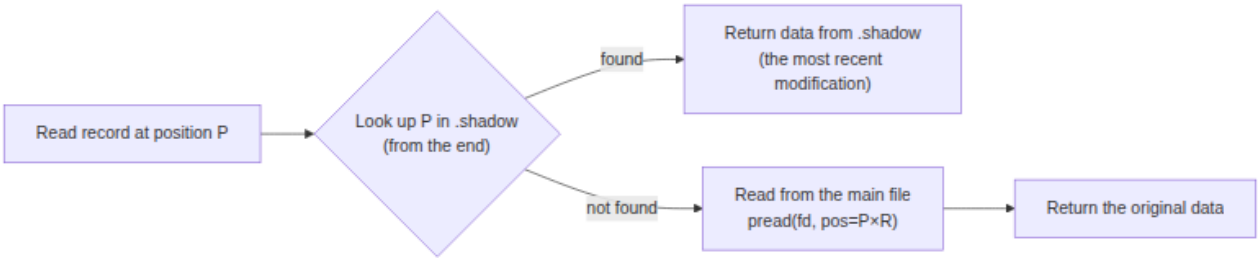



Fig. 19. Record read priority relative to the shadow file

Fig. 19 shows the record-read logic: the system first checks for an entry in `.shadow`, and only reads the record from the main file if there is none.

**Merging (merge)** The `merge()` operation merges changes from the shadow file into the main file and clears the shadow file. After merging, the original data is irrecoverably overwritten.

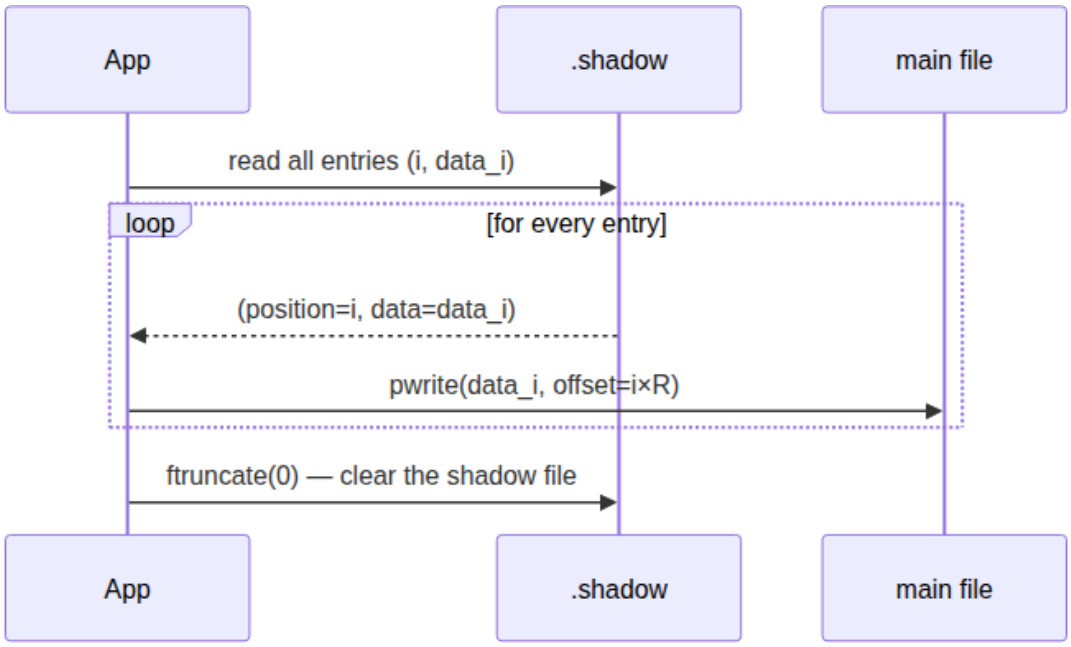



Fig. 20. Merging the shadow file into the main file

Fig. 20 shows the flow of `merge()`: successive `(position, data)` entries from `.shadow` are written to the main file, and once finished, the shadow file is cleared.

**Example: modifying a record**

```
### Stream str1: 2 INTEGER fields (4B each), recordSize = 8B
### Record 2 (original): [100, 200]
### Modification: field 0 → 999

### The .shadow file after the modification:
### offset 0: [position=2 (8B)][999, 200 (8B)]
```

Reading record 2 will return `[999, 200]`. Reading records 0 and 1 will return data from the main file (they have no entries in the shadow file).

---

### The index shadow file (.meta.shadow)

The `.meta.shadow` file is the counterpart of `.shadow` at the null-index level. It records overrides of null patterns for individual records without modifying the main `.meta` file, keeping the pairing consistent: `main file ↔ .meta` and `shadow file ↔ .meta.shadow`.

**When it's created** The `.meta.shadow` file is created automatically by `metaDataStream` when two conditions are met:

1. The store is of type `DEFAULT` or `POSIXSHD` — i.e. one that keeps record modifications in a `.shadow` file (not in the main file).

2. At least one modification of an existing record (`storage::write()` at an index other than the maximum) is made during the given session.

Condition 1 is checked during `storage::attachStorage()` — if it holds, `metaDataStream::setShadow` is called.

**File format** The `.meta.shadow` file has no header. It is a sequence of entries in the same binary format as the entries in the `.meta` file, with the difference that the `recordCount` field stores the **absolute record index** (not the number of records in an RLE run):

| Field | Size | Meaning in `.meta.shadow` |
|---|---|---|
| `gapFlag` | 1 B | always 0 (overrides are never gaps) |
| `recordCount` | 8 B (size_t) | absolute index of the overridden record |
| `bitsetSize` | 8 B (size_t) | number of descriptor fields (N) |
| `bitset` | ⌈N/8⌉ B | the new null pattern for this record |

Every call to `onRecordModified()` in shadow mode appends one entry to the end of the file. Multiple entries for the same position are allowed — the **last** entry governs (last-write-wins semantics, matching the `.shadow` file).

**Read priority** In shadow mode, `getNullBitset(i)` scans the list of overrides from the end. If it finds an entry for index `i`, it returns that entry's null pattern without consulting the main index (Fig. 21):

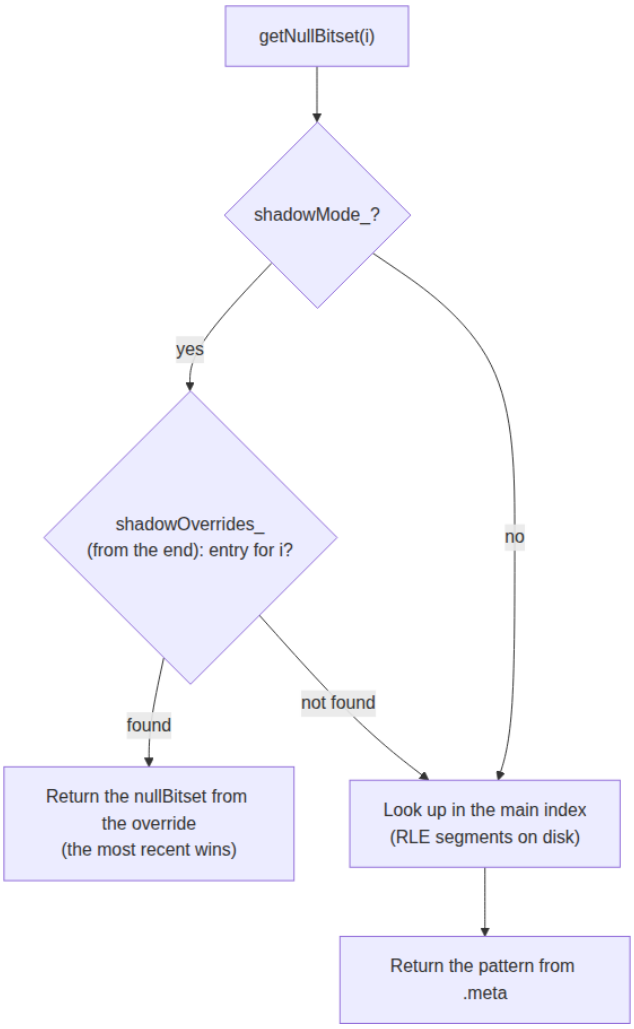



Fig. 21. Null-pattern read priority — main index vs. index shadow

**Lifecycle** The `.meta.shadow` file is managed in parallel with the data shadow file:

| Event on the `.shadow` file | Action on `.meta.shadow` |
|---|---|
| First record modification | File creation; first entry appended |
| Subsequent modifications | Further entries appended |
| `merge()` — merging the shadow into the main file | `mergeShadow()` — overrides applied to `.meta`; file deleted |
| `purge()` / `reset()` — discarding the shadow | `discardShadow()` — file deleted without merging |
| Process restart | `setShadowMode(true)` → `loadShadow()` — file read; overrides restored in memory |
| Removal of a temporary store (destructor) | `.meta.shadow` file deleted along with `.meta` |

**Persistence across restarts** After the process restarts, a new `metaDataStream` object restores the shadow state via `loadShadow()` (Fig. 22):

1. It reads all entries from `.meta.shadow` (no header — a direct format).
2. It loads them into `shadowOverrides_` in write order.
3. `getNullBitset()` and subsequent calls to `onRecordModified()` behave exactly as they did before the restart.

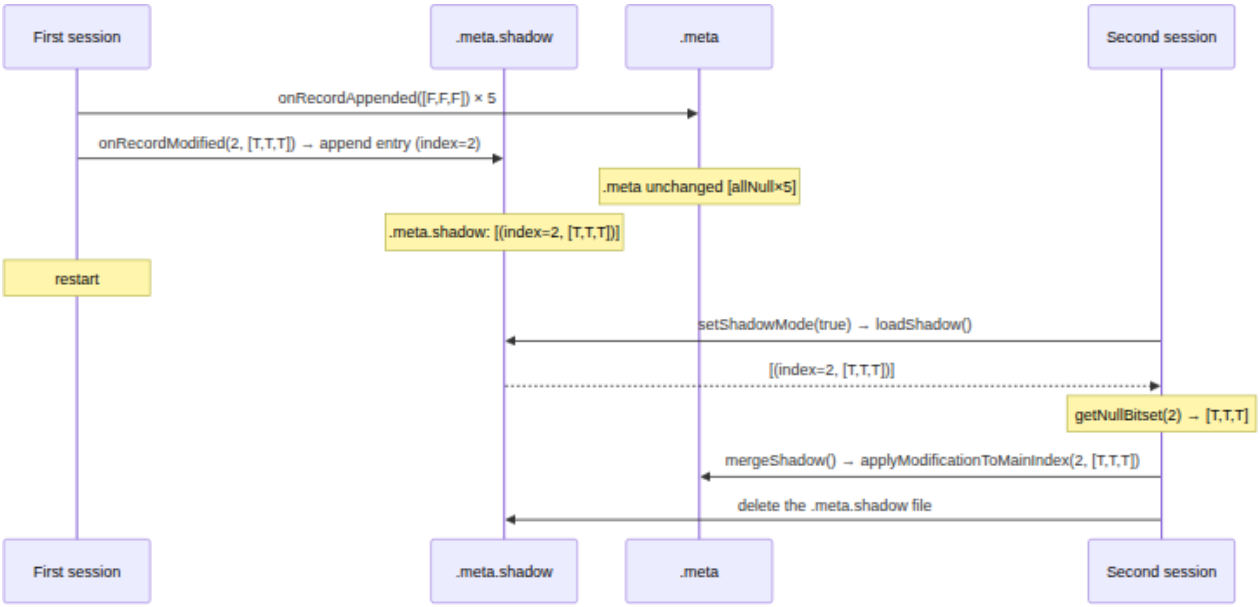



Fig. 22. Index shadow — restoring null patterns after a restart

**Usage example — correcting a record while preserving consistency**

```
### 5 records in stream str1, 3 FLOAT fields
### Record 2 has a null value in field 0: nullBitset=[T,F,F]
### The operator corrects field 0 of record 2 → pattern changes to [F,F,F]

### Operations:
storage.write(rec2_corrected, pos=2)
  → .shadow: append (position=2, data_corrected)
  → metaDataStream.onRecordModified(2, [F,F,F])
    → shadow mode: .meta.shadow: append (index=2, [F,F,F])

### File state:
### .meta        – unchanged: [isGap=F, count=2, [F,F,F]],
### >> [isGap=F, count=1, [T,F,F]], [isGap=F, count=2, [F,F,F]]
### .meta.shadow – new entry: [gapFlag=0, recordCount=2, bitset=[F,F,F]]

### Read:
getNullBitset(2) → [F,F,F]  (from .meta.shadow)
getNullBitset(1) → [F,F,F]  (from .meta)
```

```
### After merging:
storage.merge() → .shadow absorbed into the main file
metaDataStream.mergeShadow() → .meta rebuilt, .meta.shadow deleted
### .meta after merge: [isGap=F, count=5, [F,F,F]]  (all records complete)
```

**Note**

The `.meta.shadow` mechanism is tested in the unit test `index_shadow_scenario` (`test_metaDataStream_usage.cpp`).

---

### The relationship between the files

In this section, the relationships between the files are shown at two levels. The structural level describes how the data file carries the records, the `.desc` descriptor defines their format, the `.meta` file stores information about null values and transmission gaps, `.shadow` collects data modifications without destroying the original, and `.meta.shadow` similarly collects overrides of null patterns. The operational level (Fig. 23) shows the read and write flow: reads check `.shadow` and `.meta.shadow` first, `merge()` moves corrections into the main file and the main index, and the `append`, `update`, and `read` operations keep the data and metadata consistent throughout the artifact's lifecycle.

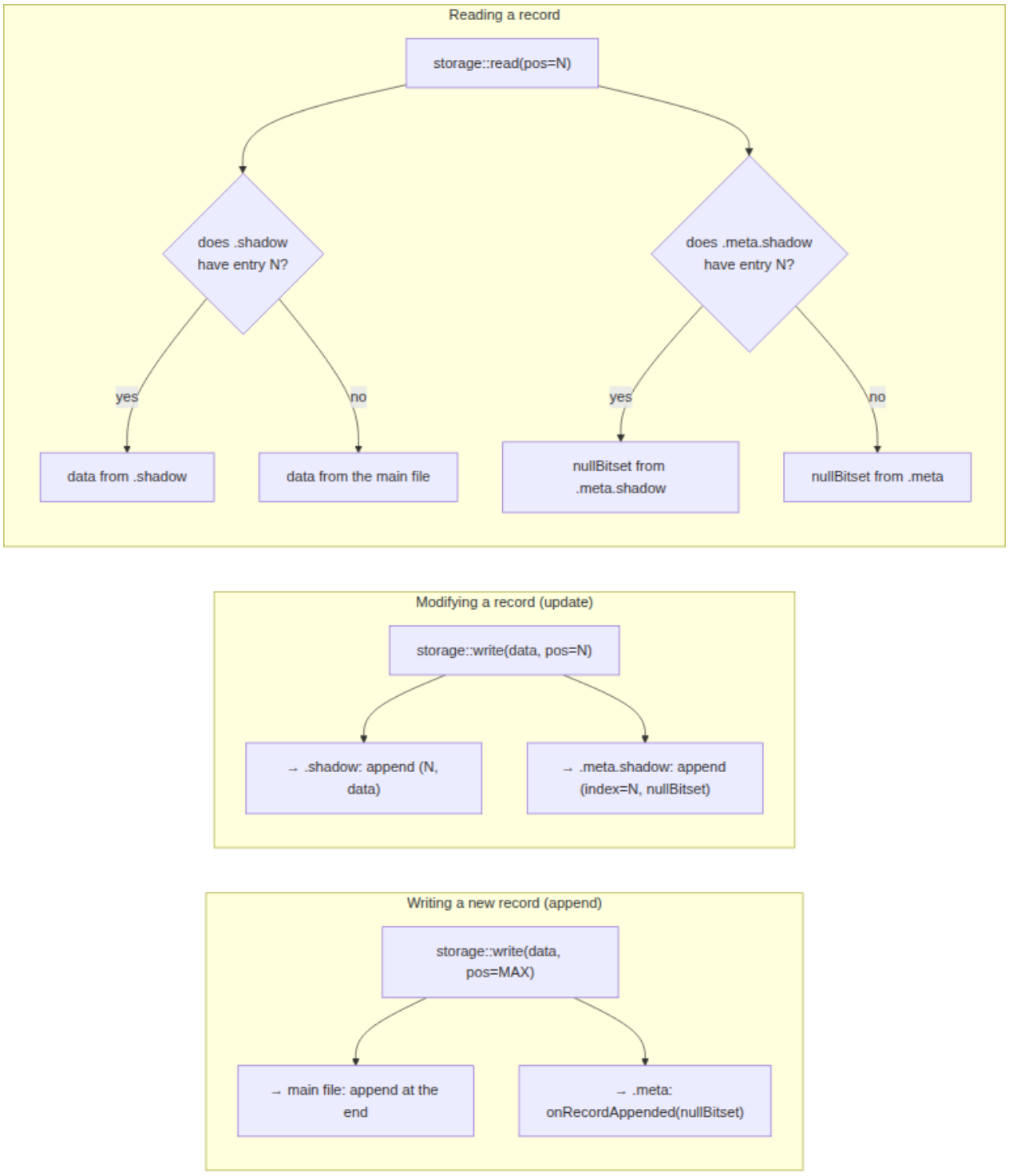



Fig. 23. The relationship between an artifact's write, modify, and read operations

Fig. 23 shows the flow of `append`, `update`, and `read` operations through the `storage` layer, and their direct effect on the data file, `.meta`, `.shadow`, and `.meta.shadow`.

### Starting point — a binary file without metadata

The simplest possible way to record a time series is a sequence of raw values in a binary file: fixed record size, no header, no structure description. This approach has one advantage — minimal overhead — and a number of significant limitations:

- Interpreting the data requires knowledge external to the file (field names, types, order).
- No information about transmission gaps — continuity of the data is only apparent.
- Every modification of a historical record irreversibly destroys the original data.
- A change to the record structure invalidates the entire file.

RetractorDB records data from sensors operating in real time, where power interruptions, signal loss, and the need for retrospective data correction are normal operational occurrences, not exceptions. The four-file structure directly addresses each of these limitations.

### What each file contributes

#### The descriptor (`.desc`) — self-description and independence from code

A binary data file is useless without knowledge of the record structure. The descriptor stores that knowledge alongside the data, which means:

- Data can be read and interpreted without access to the source code or configuration — the `.desc` file is enough.
- The `xtrdb` tool can analyze any artifact without additional parameters.
- Changes to a stream's structure (adding a field, changing a type) are explicit and versionable.
- The `TYPE` field in the descriptor determines the storage strategy, allowing the same engine to handle durable artifacts, ephemeral ephemerides, and external data sources without changing the query logic.

#### The metadata file (`.meta`) — trustworthiness of the time series

A time series with gaps, treated as continuous, leads to incorrect time-window computations, incorrect aggregations, and false correlations. The `.meta` file provides:

- The ability to distinguish a record with a zero value from a record that is absent (null) — semantically completely different states.
- Recording of transmission gaps without inserting fake records into the data file — the binary file stays dense and positionally addressable.
- RLE compression — typical time series have long stretches without nulls, so the metadata cost is close to zero for good-quality data.
- The ability to reconstruct the exact recording schedule, including gap lengths, which is necessary when computing intervals in the stream algebra.

#### The shadow file (`.shadow`) — non-destructive data correction

In measurement systems, correcting faulty samples after the fact is a standard procedure. Overwriting the binary file is irreversible and destroys the evidence of the original measurement. The shadow file:

- Lets you correct any historical record without modifying the main file.
- Preserves the original measurement as the default — deleting the `.shadow` file fully restores the initial state.
- Allows merging (`merge`) corrections into the main file only when that is a deliberate decision by the operator, not a side effect of writing.
- Separates certified data (the main file) from working data (the shadow file), which matters in applications requiring auditability.

**The index shadow file (`.meta.shadow`) — metadata consistency during correction**

A correction to a record in the data shadow file must be reflected in the null index — otherwise `getNullBitset()` would return a stale pattern from the main `.meta`. The `.meta.shadow` file:

- Maintains consistency between the pairs: `main file ↔ .meta` and `.shadow ↔ .meta.shadow`.
- Lets `getNullBitset()` return the current null pattern for a corrected record without modifying the main index.
- Tracks the lifecycle of the data shadow file — merged and deleted exactly alongside `.shadow`.
- Enables full state recovery after a restart: overrides loaded from `.meta.shadow` are immediately available without re-scanning the data shadow file.

## File Rotation Mechanism

By file rotation we mean the controlled closing of the current set of data and metadata files and moving them to historical versions (`.old<N>`), so that a new session can begin writing from a clean state without losing earlier measurements. This is done in order to separate successive acquisition sessions, preserve a full audit trail, and make it easier to diagnose problems over time. The goal of rotation is both to maintain operational tidiness (a current working set plus a session archive) and to make it possible to recover and compare historical data.

**Note**

The functionality described here is covered by the tests: `rotation_test`, `retention`, described in the appendix Integration Tests.

### Default behavior (without the `ROTATION` directive)

Without the `ROTATION` directive in the RQL script, `xretractor` **deletes** artifact files (binary data, `.desc`, `.meta`) on every startup and begins recording from scratch. Declaration files (`DECLARE`) and ephemerides are not deleted — they have no files on disk.

### The `ROTATION` directive and the session counter

The `ROTATION` directive enables history-preservation mode. It takes the path to a file that stores a persistent session counter:

```
ROTATION rdb_counter
```

The `PersistentCounter` object reads the value `N` from the file at startup (`getCount() = N`) and writes `N+1` at shutdown. The counter increases monotonically with every `xretractor` session.

### Control flow during rotation

Here we want to show the full lifecycle of the files during one session and the transition to the next. The diagram (Fig. 24) is meant to explain the order of events: detecting rotation at startup, creating a new `.meta` index, normal data writes during operation, and archiving files at process shutdown. The key takeaway is that rotation is not a single operation, but a process spread out over time, spanning the start and stop of a session.

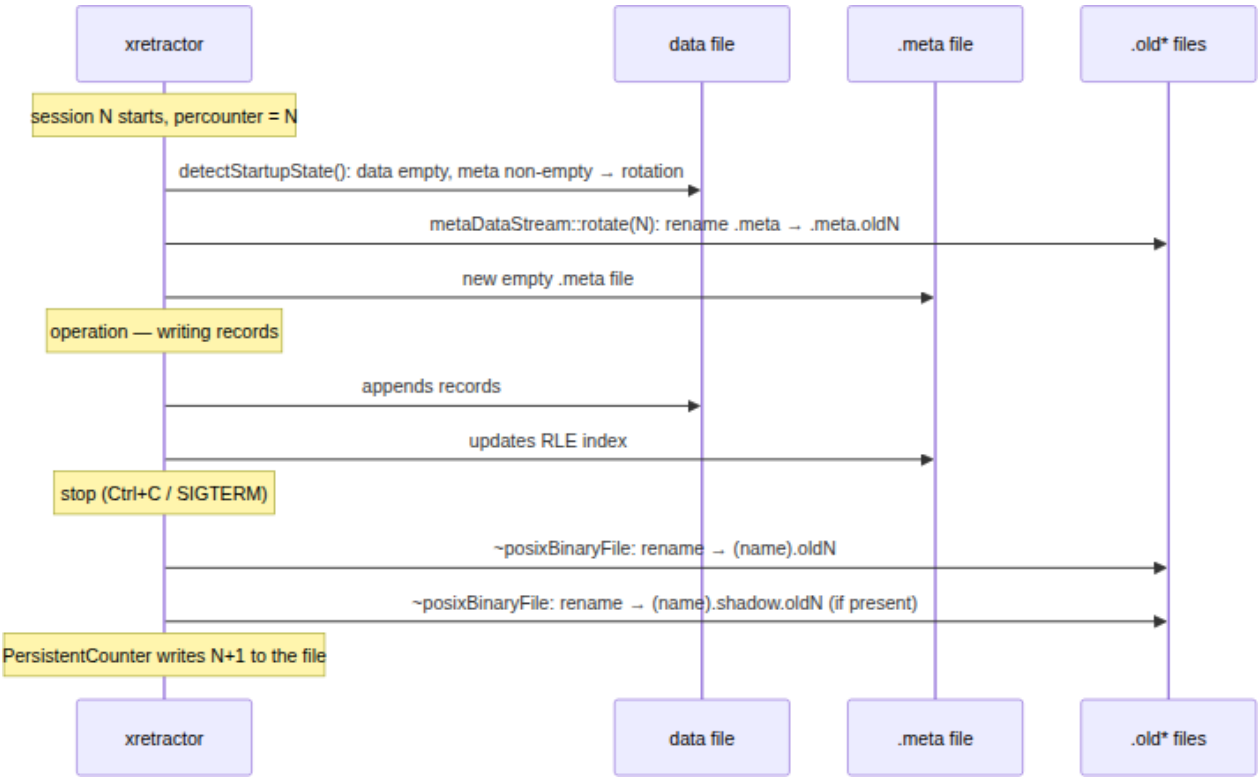



Fig. 24. File rotation sequence — session start and stop

Rotation of the `.meta` file happens **at the start** of session N — `detectStartupState()` detects the inconsistency (data file empty, index non-empty from an old session) and calls `metaDataStream::rotate(N)`. The binary data file is only renamed **at session shutdown**, by the `posixBinaryFile` destructor.

### What ends up in `.old<N>` files

| File | When it's created |
|---|---|
| `<name>.oldN` | Session N shutdown — the `posixBinaryFile` destructor renames the data file |
| `<name>.shadow.oldN` | Session N shutdown — the `posixBinaryFileWithShadow` destructor renames the shadow file |
| `<name>.meta.oldN` | Session N startup — `detectStartupState()` detects rotation and renames the `.meta` left behind by session N−1 |

As a consequence of this ordering, there is an offset of 1: the file `.meta.oldN` contains the null metadata for the data from session `N−1`, while `.oldN` contains the data from session `N`. In the `ROTATED FILES` section of the `xtrdb -s` tool, files are grouped by their numeric suffix — so the `.oldN` and `.meta.oldN` pairs differ by 1 relative to the session they physically correspond to.

### Example: sequence of three sessions

After three completed sessions (0, 1, 2), and during a fourth (3):

```
measurement.old0         ← data from session 0 (written during session 0,
                           renamed by session 0's destructor)
measurement.meta.old1    ← metadata from session 0 (renamed at the start of session 1)
measurement.old1         ← data from session 1
measurement.meta.old2    ← metadata from session 1 (renamed at the start of session 2)
measurement.old2         ← data from session 2
measurement.meta.old3    ← metadata from session 2 (renamed at the start of session 3)
measurement              ← current data (session 3)
measurement.meta         ← current metadata (session 3)
```

The `xtrdb -s` view during session 3:

```
$ xtrdb -s measurement
...
┌──────────────────────────────────────────────────────────────────────┐
│  ROTATED FILES                                                       │
│  [3] measurement.meta.old3                                  26 B     │
│  [2] measurement.old2                                       800 B    │
│      measurement.meta.old2                                   26 B    │
│  [1] measurement.old1                                       800 B    │
│      measurement.meta.old1                                   26 B    │
│  [0] measurement.old0                                       400 B    │
└──────────────────────────────────────────────────────────────────────┘
```

The file `measurement.meta.old3` stands alone in group `[3]` — the corresponding file `measurement.old3` will only be created once the current session is closed.

### Opening a rotated file in `xtrdb`

Rotated files can be examined with the `open` command in `xtrdb`'s interactive mode. The `open` command automatically strips the base name (removes `.old<N>`) and looks for the descriptor `<base_name>.desc`:

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

## Inspection Tool: `xtrdb -s`

The command `xtrdb -s <path>` displays a complete picture of an artifact's storage state — without starting the `xretractor` process, without entering interactive mode. Just point it at the base path (without extension), and the tool finds the associated files on its own: `.desc`, binary data, `.meta`, `.shadow`, cyclic segments, and rotated files.

**Note**

The functionality described here is covered by the test: `issue153_storagemap_meta_cases`, described in the appendix Integration Tests.

### Purpose and use cases

| Situation | What `xtrdb -s` gives you |
|---|---|
| Post-crash diagnosis | You can immediately see whether the data file is consistent with the metadata — differing record counts signal a problem |
| Retention verification | The DATA TOTAL section shows the segment breakdown and the current fill level of the circular buffer |
| Modification control | The SHADOW section reveals the number of uncommitted changes — `Updates: N` means `merge()` has not been run |
| Data-quality analysis | The META bar, with the symbols `=`, `-`, `~`, `X`, shows the null/gap pattern without parsing the binary file |
| Rotation-history audit | The ROTATED FILES section lists old versions of the file after successive rotations |

The command is **read-only** — it does not modify any file. It can also be run while `xretractor` is not running.

### What the map shows

The whole report is framed with box-drawing characters. The top part is a three-column overview map:

```
┌──────────────────────────────────────────────────────────────┐
│   Storage map: <name>                                        │
├──────────────┬──────────────┬────────────────────────────────┤
│ [shadow]     │ [binary data] │ [meta index]                  │
├──────────────┼──────────────┼────────────────────────────────┤
│ ...          │ ...          │ ...                           │
├──────────────┴──────────────┴────────────────────────────────┤
│   SECTION  ...                                               │
└──────────────────────────────────────────────────────────────┘
```

Each row of the map corresponds to one RLE segment or one data segment:

| Column | Content |
|---|---|
| `[shadow]` | For an artifact without retention: the number of unwritten modifications (`N updates`). For segmented retention: the segment label `sN` with its modification count. |
| `[binary data]` | The record-index range in the binary file (`begin-end`), or the segment label `sN begin-end`. Rows for a transmission gap have this field empty. |
| `[meta index]` | Description of the RLE segment from the `.meta` file: number of records and the null pattern in the form `[====]`. |

Below the map come further sections:

| Section | Description |
|---|---|
| `DESCRIPTOR` | Path and size of the `.desc` file, the field list with types and sizes, the record size in bytes. |
| `DATA` | Number of records, path to the data file. With retention (`RETENTION`): the segment breakdown, the policy (segment count and capacity), the maximum allowed buffer size, the list of `_segment_*` files. |
| `META` | Number of RLE segments and records in the index, a graphical bar showing the null pattern over time. |
| `SHADOW` | Path and size of the shadow file, and the number of uncommitted modifications. |
| `ROTATED FILES` | Files from previous rotations (`.old1`, `.old2`, ...) with their sizes. |

**META bar legend**

```
[====] — data with no null values
[----] — partial nulls (at least one field is null)
[~~~~] — all fields are null (nullfill)
[XXXX] — transmission gap
```

### Example 1 — a simple artifact

Stream `measurement` with two fields, 100 records, no modifications, no gaps:

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

```
$ xtrdb -s measurement
```

```
┌──────────────────────────────────────────────────────────────────┐
│   Storage map: measurement                                       │
├──────────────────────────────────────────────────────────────────┤
│ [shadow]   │ [binary data] │ [meta index]                        │
├────────────┼───────────────┼─────────────────────────────────────┤
│            │ 0-100         │ [====] 100 records, no nulls        │
├────────────┴───────────────┴─────────────────────────────────────┤
│   DESCRIPTOR  measurement.desc                          43 B     │
│   INTEGER  ts                                            4 B     │
│   FLOAT  value                                           4 B     │
│   Record size:                                           8 B     │
├──────────────────────────────────────────────────────────────────┤
│   DATA        measurement                              800 B     │
│   Records: 100                                                   │
├──────────────────────────────────────────────────────────────────┤
│   META        measurement.meta                          26 B     │
│   Segments: 1   Records: 100                                     │
│   [=========================100==========================]       │
│   Legend: [====] data  [----] partial null                       │
│           [~~~~] nullfill  [XXXX] gap                            │
├──────────────────────────────────────────────────────────────────┤
│   SHADOW      measurement.shadow (missing)                0 B    │
└──────────────────────────────────────────────────────────────────┘
```

Interpretation: one RLE segment, no gaps, no nulls, no shadow file present. The binary file is exactly 100 × 8 = 800 bytes.

---

### Example 2 — an artifact with a transmission gap and a modification

Stream `sensor` with three fields. After 50 records there was a gap (10 interval units), then 30 records arrived with partial gaps in the `pressure` field. Two records were later modified (a shadow file is present):

```
{
  INTEGER  ts
```

```
  FLOAT    temp
  FLOAT    pressure
  TYPE     DEFAULT
}
```

```
$ xtrdb -s sensor
```

```
┌──────────────────────────────────────────────────────────────────┐
│   Storage map: sensor                                            │
├──────────────┬──────────────┬────────────────────────────────────┤
│ [shadow]     │ [binary data] │ [meta index]                      │
├──────────────┼──────────────┼────────────────────────────────────┤
│              │ 0-50         │ [====] 50 records, no nulls       │
│              │              │ [XXXX] 10 records, gap            │
│ 2 updates    │ 50-80        │ [----] 30 records, some nulls     │
├──────────────┴──────────────┴────────────────────────────────────┤
│   DESCRIPTOR  sensor.desc                              52 B      │
│   INTEGER  ts                                           4 B      │
│   FLOAT  temp                                           4 B      │
│   FLOAT  pressure                                       4 B      │
│   Record size:                                         12 B      │
├──────────────────────────────────────────────────────────────────┤
│   DATA        sensor                                  960 B      │
│   Records: 80                                                    │
├──────────────────────────────────────────────────────────────────┤
│   META        sensor.meta                               60 B     │
│   Segments: 3   Records: 80                                      │
│   [===========50===========][XXgap:10XX][-------30---------]    │
│   Legend: [====] data  [----] partial null                       │
│           [~~~~] nullfill  [XXXX] gap                            │
├──────────────────────────────────────────────────────────────────┤
│   SHADOW      sensor.shadow                             26 B     │
│   Updates: 2                                                     │
└──────────────────────────────────────────────────────────────────┘
```

Interpretation: the binary file contains 80 records (the gap takes no space in the data file); the gap is encoded solely in `.meta`. The `[binary data]` column shows an empty range for the gap segment — there is no binary data for it. The `pressure` field in records 50–79 has null values in some records (`[----]`). The shadow file contains 2 modifications that have not yet been merged into the main file.

---

**Example 3 — an artifact with segmented retention**

Stream `buffer` with cyclic retention: up to 10 segments of 100 records each (1000 records total). Currently 280 records have been written, across three segments:

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

```
$ xtrdb -s buffer
```

```
┌──────────────────────────────────────────────────────────────────┐
│   Storage map: buffer                                            │
├──────────────────────────────────────────────────────────────────┤
│ [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  buffer.desc                              48 B      │
│   DOUBLE  value                                        8 B       │
│   Record size:                                         8 B       │
├──────────────────────────────────────────────────────────────────┤
│   DATA TOTAL  rec=280 src=0 seg=280                  2240 B      │
│   Records: 280                                                   │
│   Source: buffer   Segments: buffer_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] buffer_segment_0 rec:100 range:0-100                     │
│     [1] buffer_segment_1 rec:100 range:100-200                   │
│     [2] buffer_segment_2 rec:80 range:200-280                    │
├──────────────────────────────────────────────────────────────────┤
│   META        buffer.meta                              26 B      │
│   Segments: 1   Records: 280                                     │
│   [=========================280==========================]       │
│   Legend: [====] data  [----] partial null                       │
│           [~~~~] nullfill  [XXXX] gap                            │
├──────────────────────────────────────────────────────────────────┤
│   SHADOW      buffer.shadow (missing)                  0 B       │
└──────────────────────────────────────────────────────────────────┘
```

Interpretation: the `[binary data]` column shows each segment with its `sN` label and its global index range. The DATA TOTAL section gives a full breakdown: `src=0` (no records outside the segments), `seg=280` (all records within segments). Once the buffer

fills up (10 segments × 100 = 1000 records), the oldest segment will be deleted and a new one appended.

## Summary: Rationale for the Chosen Structure

This chapter draws together the conclusions from every part of the data-storage-format documentation and explains why the adopted four-file structure is minimal and sufficient for a real-time time-series recording system.

### The file set and accessor types

Every artifact or substrate consists of up to four files — the binary data file, the `.desc` descriptor, the `.meta` index, and the `.shadow` file. The `TYPE` field in the descriptor selects the `FileInterface` implementation: `DEFAULT` (data + shadow + retention), `MEMORY` (RAM only, ephemerides), `DEVICE` / `TEXTSOURCE` (read-only external sources), and intermediate variants. The accessor is chosen once, when `storage` is initialized — the RQL query logic knows nothing about storage details.

### Artifact files

**The descriptor (`.desc`)** defines the record schema in ANTLR4 grammar: field names, types (`BYTE`, `INTEGER`, `FLOAT`, `DOUBLE`, `RATIONAL`, `STRING`), array sizes, retention policy (`RETENTION`, `RETMEMORY`), and accessor type (`TYPE`). The record size `R` is the sum, in bytes, of all data fields — the meta-descriptor's own fields take no space in the record. Having the descriptor alongside the data means self-description: the `xtrdb` tool, or any code, can interpret an artifact without access to the source code.

**The binary data file** is a flat sequence of fixed-length `R`-byte records with no header. Record `i` always sits at offset `i × R`. An `append` operation writes to the end; an `update` operation — when a `.shadow` file is present — goes to the shadow file, rather than overwriting the main file.

**The metadata file (`.meta`)** stores a compressed RLE index of null values and transmission gaps. Each RLE entry describes a run of consecutive records sharing an identical null pattern: an `isGap` flag, a `recordCount`, the bitset size, and the bitset itself. A transmission gap (`gap`) exists only in `.meta` — the binary file does not record it and stays dense. The managing class is `rdb::metaDataStream`: it buffers the current segment in `currentEntry_`, writes a segment to disk only when the pattern changes, and the `tailDirty_` mechanism ensures the file size does not grow under continuous, uniform data arrival. After a restart, `loadIndex()` restores the state and brings the last non-gap segment back into memory, allowing the RLE run to continue.

**The shadow file (`.shadow`)** collects record modifications as a sequence of `(position, data)` entries. Reading a record checks `.shadow` from the end (the most recent modification wins); if there's no entry, it reads from the main file. Deleting `.shadow` fully restores the original state. The `merge()` operation writes the corrections back into the main file and clears the shadow file.

### The rotation mechanism

The `ROTATION rdb_counter` directive turns on session-history preservation mode. `PersistentCounter` stores a monotonically increasing session number `N`. Rotation is a process spread out over time: at the **start** of session N, `detectStartupState()` detects an inconsistency (data file empty, `.meta` non-empty) and renames `.meta` to `.meta.oldN`; at session **shutdown**, the `posixBinaryFile` destructor renames the data file to `.oldN` and the shadow file to `.shadow.oldN`. As a consequence of this ordering there is an offset of 1: `.meta.oldN` contains the metadata for session `N−1`, while `.oldN` contains the data for session `N`. Without the `ROTATION` directive, artifact files are deleted on every startup.

### The inspection tool `xtrdb -s`

The command `xtrdb -s <path>` is the only tool for inspecting storage state without starting `xretractor`. The report consists of an overview map (columns: shadow, binary data, meta index) and detailed sections: `DESCRIPTOR`, `DATA` (or `DATA TOTAL` for segmented retention), `META` with an RLE bar, `SHADOW` with the count of uncommitted modifications, and `ROTATED FILES` with the rotation history. The META bar uses four symbols: `=` (data, no nulls), `-` (partial nulls), `~` (nullfill), `X` (gap). The tool is read-only and works while the `xretractor` process is not running.

---

### Comparison of approaches

| Property | Raw binary file | RetractorDB structure |
|---|---|---|
| Self-description | none — requires external documentation | yes — the `.desc` descriptor travels with the data |
| Transmission-gap handling | none — gaps are invisible or represented by fake records | yes — `.meta` records gaps without extending the data file |
| Per-field null values | none — zero and null are indistinguishable | yes — a null bitset in `.meta` |
| Correction of historical data | destructive | non-destructive — `.shadow` |
| Restoring the original after a correction | impossible | yes — delete `.shadow` |
| Multiple storage strategies | none | yes — the `TYPE` field in the descriptor |
| Cost for gap-free, null-free data | — | minimal: `.meta` ≈ a 17 B header + 1 RLE entry |

## 4.6. Compilation and Plan Construction

The compilation process happens before every run of the xretractor process. An argument in the form of a file with a sequence of commands and queries is required.

Based on the flow shown in Fig. 13, I prepared a description of the process in Fig. 25, showing the compilation process in development mode. The compilation process can be invoked even while another xretractor process is already running. Locking a single instance of the data-processing process applies only to the query-execution-plan process. Invoking compilation in this case, even if that process is already running in the system, will not report an error. Attempting to start another processing run — it will.

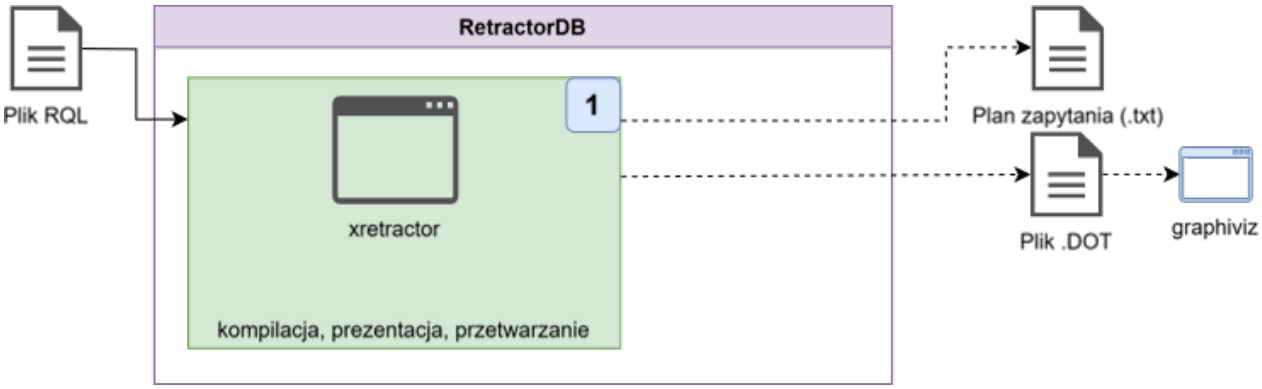



Fig. 25. The compilation process

As an example file for compilation, we'll use a file query.rql with the following content:

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

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

This is a very simple example of a file containing two directives. The first declares the existence of an ephemeris in the form of a binary data source containing 4-byte INTEGER values. Data from this file will be read at a rate of 10 times per second. And the name of this object is core0.

The second command creates an artifact named str1, taking ephemeral data shifted in time by two reads, i.e. 0.2 seconds. While building successive elements of the output stream, the data read from core0 is processed, and the value 1 is added to every value read.

To compile this file, the following command must be invoked:

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

The following system response will be printed on the screen:

```
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
```

Omitting the -c parameter will cause the system to attempt compilation and immediately send the compiled query execution plan for execution. This will cause an error, since the data file datafile1.dat presumably hasn't been prepared yet.

Besides the text view, we can also look at the compilation output in graphical form. To do this, invoke the following sequence of commands:

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

Assuming you have the dot program from the graphviz package installed in your runtime environment, this command will generate an image file showing the system's response in the form of a graph.

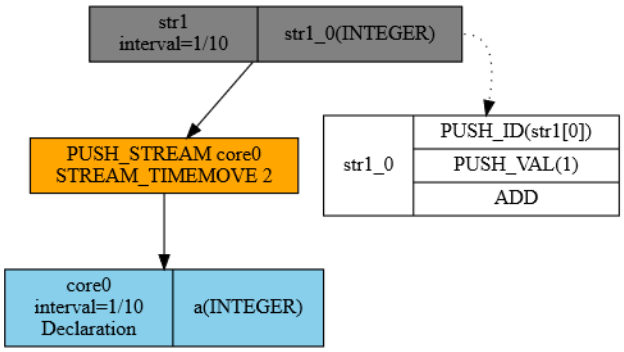



Fig. 26. Graphical representation of a query plan

RetractorDB can generate an image in response to one of the requested data-processing chains. The graphical presentation is most suitable for creating and presenting data-processing graphs. Unfortunately, readability suffers for very complex schemas.

Fig. 26 shows the trivial query execution plan produced by compiling the two-line query.rql file. At the very top we see the object str1, producing artifacts at a rate of 10 records per second. Information about the artifact-creation rate does not appear in the query; it is computed based on the algebraic expression in the FROM clause of the SELECT query. We can also see how successive records of the str1 stream are produced. Here we have a typical stack-based data-processing algorithm. First, the ephemeral value produced by the algebraic expression is pushed onto the stack, then the value 1 is placed on the stack. The ADD instruction pops both values off the stack, leaving the sum on the stack. What remains on the stack — i.e. the result of the addition — is placed into the field of the record being created.

On the other side we see stream operations. Stream operations are carried out in

a different domain. There, we process objects of one or two values. Operations act either on two streams, or on a single stream with an argument. The classic stack has no application for algebraic stream operations. For simplicity, the notation resembles stack operations somewhat. We see, in the attached example, that operations on current data are carried out by shifting the data in time by 2. I deliberately do not say that this is 2 seconds — here 2 denotes a relative value with respect to the arrival rate. For an arrival rate of 10 samples per second, the value 2 means a time shift of 0.2 seconds.

Complex algebraic expressions involving at least two stream operators give rise to the substrates mentioned in previous chapters. Every query whose FROM-clause algebraic expression contains more than one operator is broken down into interdependent two-argument operations. The substrate's argument list is, by default, the full expansion of the schema.

### Available xretractor flags

Different sets of flags are available in compile mode (`-c`) and in execution mode. Below are the compile-mode flags used for generating graphs:

| Flag | Full name | Meaning |
|---|---|---|
| `-c` | `--onlycompile` | compile only — does not start processing |
| `-d` | `--dot` | generate output in DOT (graphviz) format |
| `-f` | `--fields` | show stream fields in the DOT graph |
| `-s` | `--streamprogs` | show stream programs in the DOT graph |
| `-u` | `--rules` | show RULE rules in the DOT graph |
| `-p` | `--transparent` | transparent background for the DOT graph |
| `-i` | `--hideruleprog` | hide the rule-condition program (with `-u`) |
| `-m` | `--csv` | output in CSV format |

Execution-mode flags (without `-c`):

| Flag | Full name | Meaning |
|---|---|---|
| `-m N` | `--tlimitqry N` | run N processing cycles, then exit |
| `-k` | `--noanykey` | don't wait for a keypress — daemon/script mode |
| `-t` | `--realtime` | real-time mode (SCHED_FIFO, mlockall) |
| `-x` | `--xqrywait` | wait for the first xqry connection before starting |
| `-s` | `--status` | check whether an xretractor instance is already running |
| `-v` | `--verbose` | print stream parameters at startup |

**Info**

The `-m N` parameter counts iterations of the main loop, not seconds. For streams with a 0.1 s interval (10 Hz), `-m 10` means ~1 second of processing.

**Warning**

When using `-m N` in scripts and tests, always add `-x` (`--xqrywait`). Without this flag, the server may process all N cycles before the client (`xqry`) manages to connect — the client will receive no data and will wait until it times out. The `-x` flag holds off processing until the first command arrives from `xqry`.

A full list of all options with a description of each — including the `--realtime` option, which requires system privileges — can be found in Appendix A.

## 4.7. Data Processing and Distribution

Starting the data-processing process and analyzing the diagram shown in Fig. 13, we can identify the following flow — Fig. 27:

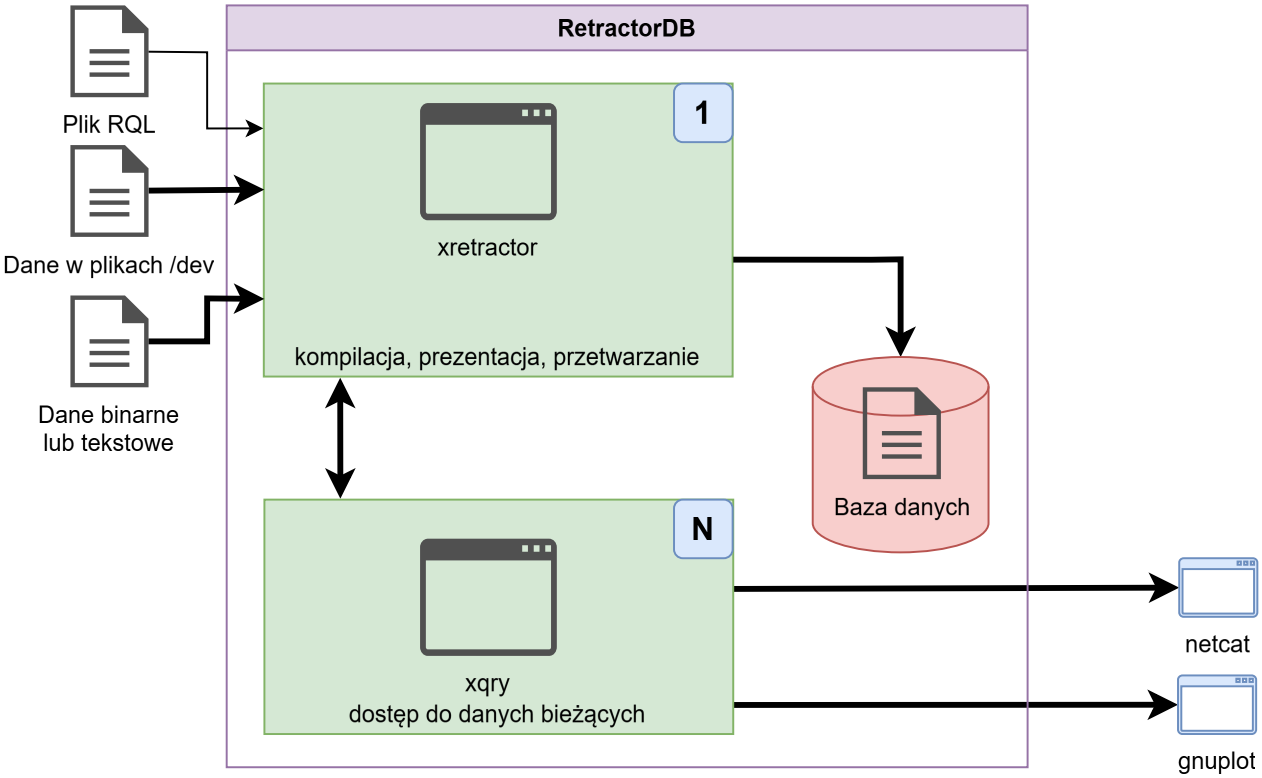



Fig. 27. Control-flow diagram of the processing workflow

To carry out the processing, we'll need to prepare data and build a data-processing chain. As input to this chain we'll use a prepared query-plan file, and we'll prepare a binary file with data. We'll build a process that processes the data and displays the results.

We'll change the source query.rql file to the following:

```
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
```

In this example we declare the existence of a text file containing text data. I suggest filling the file datafile1.txt with the following content:

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

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

The file will contain consecutive numbers from 20 to 28.

A look at the query execution plan gives us the picture in Fig. 28:

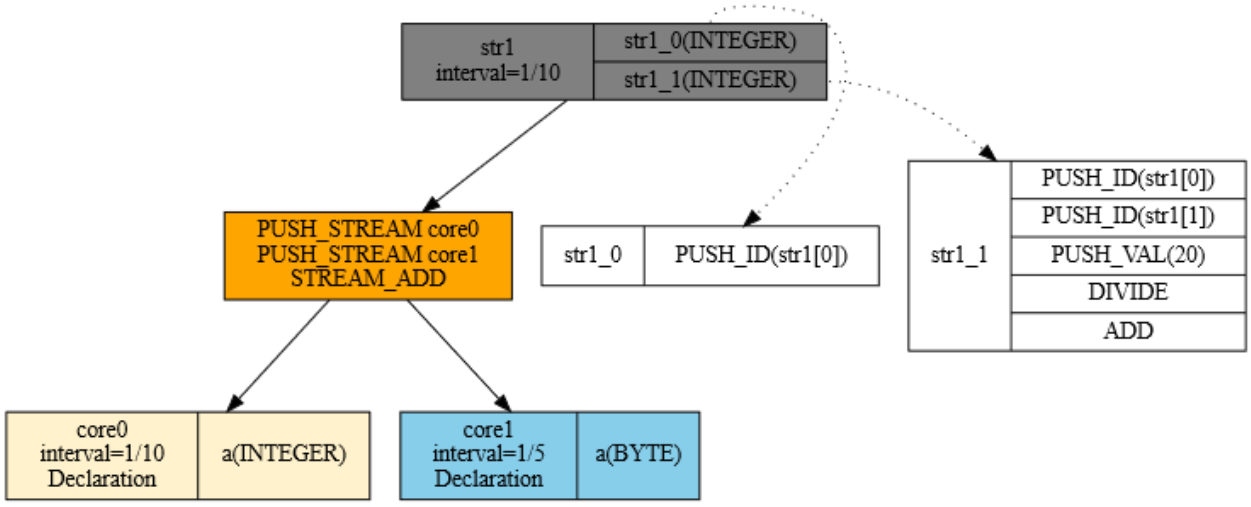



Fig. 28. Graphical representation of query plan 2

Once we've prepared the data file, we can start the compilation and data-processing

process. We do this by running the following command:

```
$ xretractor query.rql
```

And here an important property of the system comes into play. The system should immediately begin executing the process. Any key pressed in the terminal will interrupt this process.

I suggest opening a second terminal window and continuing the session there. In the second terminal window we can run the following command:

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

Something similar should appear. Of course, the counters for str1 should differ. It's logical that with every read we'll get larger values for the accumulated size of the str1 stream.

If we want to see, on screen, what's happening inside the data-processing process right now, I suggest issuing the command below, and after a few lines appear on screen, pressing any key to interrupt the process:

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

The first column contains the sequence of numbers — exactly as we entered them into the datafile1.txt file. The second column contains the result of processing. A value taken from the pseudo-random number generator, divided by 20, is added — the second column flows along beneath the first.

How can we see this graphically? I suggest running the following command:

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

The following window will appear on screen, with data streaming in live:

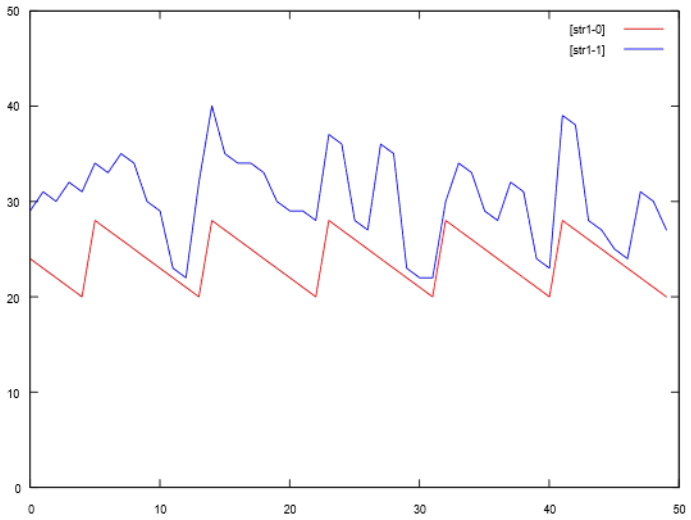



Fig. 29. Snapshot of the gnuplot window showing incoming data

In Fig. 29 we see what the data represented numerically looks like. The sawtooth shape is the first column; the irregular shape wrapping around the sawtooth is the second column. The figure shows a static snapshot — but in the actual window, this data streams in and the picture updates continuously.

A typical way to send data outside the machine on which xretractor and xqry are running is to use the command:

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

on the second computer, you need to write:

```
$ nc server_name_or_ip 8888
```

**Info**

The `-p` flag in netcat (BSD syntax) is not supported by the GNU netcat available on modern Ubuntu/Debian systems. The correct syntax is `nc -l 8888` (without `-p`).

Data transmission will take place over the network.

If we want to stop the xretractor process using the xqry command, we can run the following command:

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

After issuing this command, the xretractor process will shut down and interrupt the query plans being processed.

A recording of the process shown on screen (Fig. 30) looks as follows:

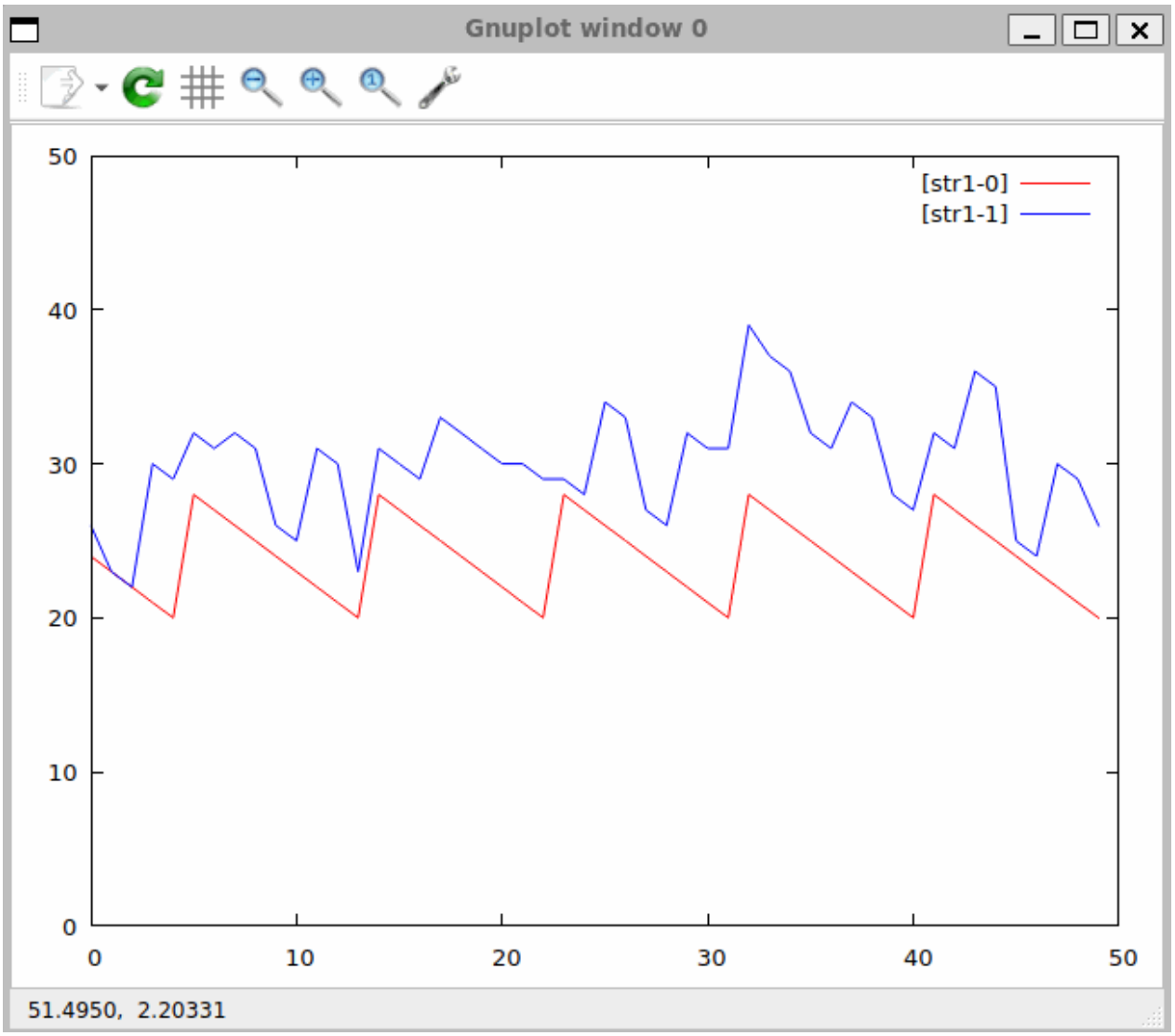



Fig. 30. Recording of real-time data processing

## 4.8. Artifact Analysis

Looking more broadly at the potential data paths in Fig. 13, the last undescribed path is the one involving the xtrdb tool.

While building the system, I needed a tool for accessing artifacts in order to run integration tests. To verify correctness, I had to compare processing results at various stages. Fig. 31 shows the complete data flow, including the role of the xtrdb tool.

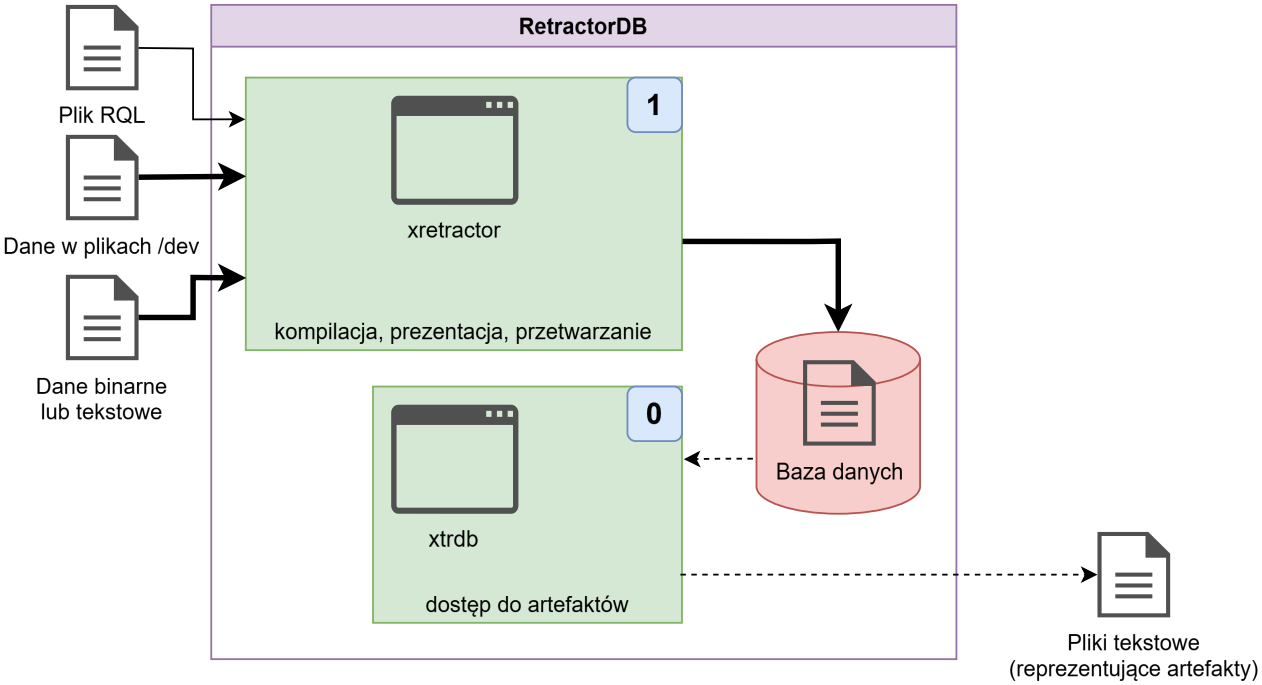



Fig. 31. Data flow in artifact analysis

To present the artifact-analysis process, the entire processing chain needs to be taken into account. We'll use the same query as before. However, we'll run our data-processing process a bit differently this time.

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

A query-processing run invoked this way will finish after 10 processing cycles. The `-m` parameter specifies the number of iterations of the main loop, not the number of seconds — the running time depends on the interval of the source streams. For streams with a 0.1 s interval (10 Hz), this means ~1 second of runtime. After it finishes, and we look at the directory in which we ran the query, we should see the following files:

```
$ 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
```

As you can see, three .desc files were created, plus one file containing artifacts. If we look inside the str1 file, we'll see fairly modest content:

```
$ 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
```

Alongside the artifact file, metadata files are also created. Their content describes the structure of the file.

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

The descriptions of the ephemeris files are far more interesting. The description files for ephemeral data point to files in the Linux filesystem.

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

Metadata description files are created automatically the moment an object is registered in RetractorDB. Remember to delete these descriptors if you modify the query.rql file.

Once the xtrdb program is started in a terminal, the tool displays a dot (`.`) as its prompt. This character is just a prompt — it is not part of the command. You can start interacting with the tool right away. Example session:

```
$ xtrdb
.open str1
ok
.desc
{       INTEGER str1_0
        INTEGER str1_1
}
.list 1
{ str1_0:20 str1_1:21 }
.quit
```

Working with this tool feels like working with a classic, old-school dbase database. There’s no state machine here, no loops or conditions — just reading and modifying

binary files described by metadata.

The main goal of this tool was to support the creation of test scripts. RetractorDB is deterministic. The system has no race conditions — data that arrives on the input should always produce the same results on the output. Unless, of course, we mix in random data, as in the example shown here.

**Note**

The functionality described here is covered by the tests: `issue113_meta_xtrdb`, `issue113_meta`, `issue113_null_txtsrc`, `Pattern5`, described in the appendix Integration Tests.

A very useful feature of this tool is the `list` and `rlist` functions — listing the initial elements of a file, or the final elements of a file, respecting the structure described in the metadata.

```
.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 }
```

I encourage you to experiment and browse the source of this tool. It's one of the less complicated, yet very useful, parts of RetractorDB.

### Inspecting null/gap metadata

Every artifact has an associated `.meta` index file, described in detail in the chapter on the storage format. Its content can be viewed directly in xtrdb with the `meta` command:

```
.open str1
ok
.meta
record 0: count=9 gap=false nullBitset=00
```

The `gap=false` entry means there is no gap in the data; `nullBitset` shows which fields contain null values (one bit per field). Data with no gaps at all forms a single entry `count=N`, where N is the total number of records.

## 4.9. Summary

In summarizing, it's worth pointing out the scope of knowledge this chapter conveys. Here I wanted to show how the individual parts of the system can be run, and what

the command sequences look like on which we build further functionality using RetractorDB.

I tried to reduce the number of potential commands to a minimum. At present I have effectively reduced the set to 3 commands. I believe a system designed this way will be maximally useful and reasonably efficient. Complexity is a separate matter. Explaining the processing itself — the new algebra, and why a plus sign doesn't mean plus — is, in itself, difficult. I hope, though, that once a certain barrier of understanding is crossed, the rest becomes obvious. The decisions made were the result of reflection, trial, and error. I want to emphasize that, quite simply, I did not find a better method.

# Chapter 5

# Query Compilation

An attentive reader will probably notice that, in the compiled query execution plans shown in the previous chapter, certain values don't match what was written in the query.

The compiler, while building a query plan, carries out the process autonomously. Sometimes it feels like you ask for one thing and get something else — at first glance this behavior seems entirely counterintuitive. And as a user, I fundamentally have no control over it. Interestingly, the outcome of the query does correspond to what I asked for in the query. Perhaps the correct title for this chapter should be: Why does the compiler do things its own way, and claim to know better?

In this chapter I want to explain how I solved the syntactic problems I encountered while building the query language.

## 5.1. Compiler input and output

### The `.rql` file

Compiler input — text in the RetractorQL language containing `DECLARE` and `SELECT` directives. The ANTLR4 parser reads the file sequentially; a reference to a stream not yet defined earlier in the file results in a compilation error.

### The ANTLR4 parser → `qTree`

The parser builds an internal representation, `qTree`: a topologically sorted `std::vector<query>`. Each element describes one stream — its field schema, its stack-instruction sequence, its dependencies on other streams, and its time interval (delta).

### The 10 compilation stages

`qTree` passes through a chain of transformations: from breaking down FROM expressions into two-argument operations, through determining deltas and byte offsets, all

the way to semantic verification and buffer-size computation. Each stage assumes the previous one succeeded.

### Execution plan → `dataModel`

At the output of compilation, every query in `qTree` has: a field schema with types and offsets, a delta, buffer sizes, and a ready instruction sequence. `dataModel` takes over this plan and executes it cyclically in real time.

The `-c` flag stops `xretractor` after this step and prints the plan to standard output — without starting processing.

## 5.2. Overview of topics covered in this chapter

The chapter is structured following the order of the compiler's stages — from a description of the data structure and the chain of stages, through the individual transformations, to error handling.

**Compilation Passes** describes the entire chain of stages in the `compiler::compile()` function. Compilation is not a single step — it's a sequence of ten successive transformations of the internal `qTree` representation, from reducing FROM expressions to two-argument form, through determining field intervals and offsets, all the way to semantic verification and buffer allocation. Each stage assumes the previous one succeeded, and returns an error message when its conditions aren't met.

**Dependency Tree Construction** describes the DAG structure produced during compilation — the foundation on which every stage rests. The roots are ephemeris declarations (external sources); inside the graph lie intermediate substrates; and the leaves are artifacts. The `-d` flag generates output in DOT format, which `graphviz` turns into a visual dependency graph. The order of queries in the `.rql` file matters — a reference to a stream not yet defined results in an error.

**Substrates** explains the `extractIntermediateStreams` stage — the first compilation step. When a FROM expression contains more than two arguments (e.g. `(core0#core1)+core2`, `core0+core1+core2`), the compiler breaks it down into two-argument operations and creates named substrates. A later stage, `deduplicateSubstrats`, detects when a substrate is structurally identical to a user query and replaces the references — avoiding duplicate computation.

**Asterisk Expansion** explains the `expandSchemaWildcards` stage. The * symbol in a SELECT clause is replaced with the full field list derived from the source stream's schema — including fields arising from stream-sum operations. An example shows how field types determine which field ends up in which position of the resulting schema.

**Interval Resolution** describes the `resolveStreamIntervals` stage. The compiler determines the delta of every output stream from the stream-algebra equations: for the `+` operator the delta is the minimum of the inputs, for `#` it's the harmonic mean, for `@(step, window)` it's a derivative of the window size. The algorithm runs iteratively — each round resolves at least one stream, until all deltas are known.

**Loop Detection** describes the mechanism built into the `resolveStreamIntervals` stage. If the number of unresolved streams stops decreasing, no stream can obtain a delta — a sign that the dependency graph contains a cycle. Compilation ends with the error `"Circular dependency in stream definitions"`. The chapter includes an example of a cyclic query and how to fix it.

**Aliasing** describes the `resolveFieldReferences` stage. An output field can be referenced either by its index in the combined schema (`str1[1]`), or by the name of the source stream with a local index (`core1[0]`). The compiler translates both forms into the same position in the output buffer.

**Underscore Symbol Processing** describes the `expandIndexWildcards` stage — syntactic sugar for parallel operations on pairs of fields. The `_` symbol in an index causes the formula to be repeated for every pair of fields from both arguments' schemas — `core0[_] * core1[_]` with two-field schemas generates two multiplying fields for the corresponding pairs. Use case: building signal-filter queries.

**Type Promotion** defines the type-promotion rules that apply throughout the compilation chain. The result of `BYTE * INTEGER` has type `INTEGER` — the compiler determines the output field's type statically, before any data is processed. The complete type hierarchy supported by RetractorDB is also described.

**Compilation Debugging** gathers diagnostic tools in one place: the `-c` flag for plan inspection, the `-c -d -f -s` pipeline for graph visualization via `graphviz`, a table of plan-instruction meanings (PUSH_ID, PUSH_STREAM, STREAM_ADD, ...), and a catalog of common compilation errors with their causes and fixes.

## 5.3. Compilation Passes

Query compilation in RetractorDB proceeds through multiple stages. Each stage transforms the internal representation of the queries — the `qTree` tree — and passes the result to the next one. The order is strictly fixed: every stage assumes the previous one succeeded.

`qTree` is a topologically sorted `std::vector<query>` — the central data structure of the compiler and the executor. Every element of the vector corresponds to one query (`SELECT` or `DECLARE`) and stores its field schema, its stack-instruction sequence, its time interval, and references to its source streams. The topological sort guarantees that a source stream always precedes its output stream — stages can process `qTree` linearly, without backtracking.

### Running example

Throughout the chapter we follow a single query — `query.rql` — through the successive stages:

```
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
```

After passing through all the stages, `xretractor -c query.rql` prints:

```
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
```

The subchapters on substrates and the _ symbol use extended variants of the same set of declarations. For how to interpret every element of this plan, see Compilation Debugging.

## The chain of stages

The chain of stages is defined by the `compiler::compile()` function:

**extractIntermediateStreams** Reduces every FROM expression to at most a two-argument form. Complex expressions like `(core0#core1)+core2`, and chained notations without parentheses (`core0+core1+core2`, `core0#core1#core2`), require intermediate streams. This stage automatically creates substrates — see Substrates.

**expandSchemaWildcards** Expands the * symbol in a SELECT clause. Replaces it with the field list derived from the source stream's schema — see Asterisk Expansion.

**resolveStreamIntervals (← loops are detected here)** Determines the time interval (delta) of every stream based on the algebraic operators and the intervals of the input streams. An iterative algorithm — each round resolves as many streams as possible. It detects cyclic dependencies by stopping when the number of unresolved streams stops decreasing — see Interval Resolution and Loop Detection.

**deduplicateSubstrats** An optimization: if two queries use the same intermediate operation (e.g. `core0#core1`), this stage points the second query at the substrate created by the first. It avoids duplicate computation — see the example in Substrates.

**resolveFieldReferences** Turns references to fields from source schemas into indices in the output schema. Handles aliasing — turning `core0[0]` into `str1[0]`, etc. — see Aliasing.

**expandIndexWildcards** Expands the _ symbol in field indices. Repeats the formula for every matching pair of fields from the arguments' schemas — see Underscore Symbol Processing.

**localizeFieldOffsets** Converts field references (`b[x]`, `c[y]`) into indices in the flattened output schema (`merged[z]`). For ADD, the index follows from the sum of the field counts of the preceding streams; for HASH, every field gets index 0 (a single-argument schema). This stage accounts not only for direct sources, but also for transitive sources hidden behind automatic substrates.

**computeRequiredCapacities** Computes the required buffer capacities for every stream, based on schema sizes and time-window requirements.

**validateConstraints** Verifies the semantic correctness of the compiled plan: type compatibility, window sizes, availability of data sources.

**applyCapacitiesToStreams** Applies the computed capacities to the stream objects. After this stage, the plan is ready for execution by `dataModel`.

Every stage returns `"OK"` or an error message — in which case compilation stops.

## 5.4. Dependency Tree Construction

The dependency tree is the query execution plan in the form of a directed graph. It is a data structure built during compilation and modified when Ad Hoc queries are added. The roots of this graph are ephemeris declarations — declarations of every kind that create external objects, so-called data sources. Inside the graph lie artifacts and substrates. At the end of the processing chain sit the artifacts — as the chain's final results.

Such a construction is a directed graph — a graph with multiple roots and multiple terminal vertices. Inside the graph there are connecting nodes. Every node lies on a path from a root to a terminal vertex. This is best visualized with an example.

Let's start by considering the following trivial query:

```
DECLARE a UINT STREAM core0, 0.1 FILE 'datafile1.txt'
SELECT str1[0] STREAM str1 FROM core0
```

We can obtain a graph highlighting the dependencies between the individual objects as follows (Fig. 32):

```
$ xretractor -c query5.rql -d > out.dot && dot -Tsvg out.dot -o out.svg
```

For a full description of the `-d -f -s` flags and how to interpret the output — see Compilation Debugging.

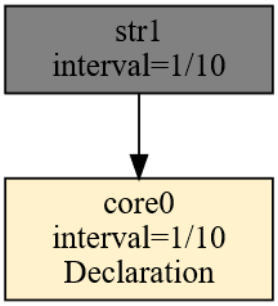



Fig. 32. Ephemeris–artifact dependency

Let's make this graph a bit more complex by adding two ephemeris declarations and an additional artifact.

```
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
```

The dependency graph for the set of queries above looks as follows (Fig. 33):

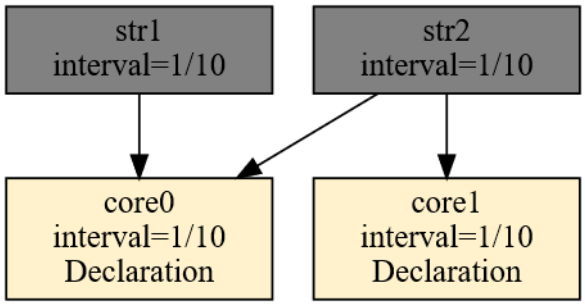



Fig. 33. Ephemerides–artifacts dependency

Let’s build an additional node that depends on artifacts. The simplest way is to add the following query at the end:

```
SELECT str3[0] STREAM str3 FROM str1#str2
```

The graph changes shape:

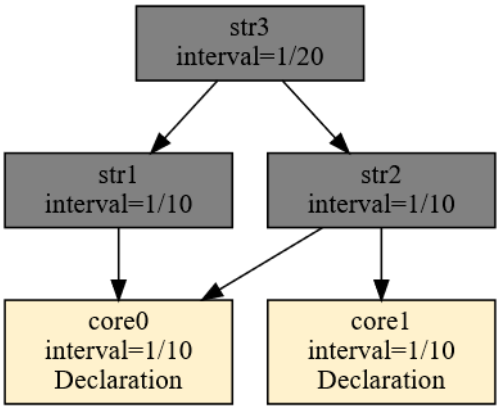



Fig. 34. Ephemerides–artifacts–artifacts dependency

As shown in Fig. 34, the str3 stream is not directly dependent on the data supplied by the core0 and core1 streams. Queries form a dependency graph, and the order in which they are invoked is well-defined. The interval value of streams grows toward the roots. This growth toward the roots follows from the interval-determination equations of the developed algebra.

Note that queries in the rql file are processed sequentially. Attempting to reference, in a query, an object that is not yet defined results in a compilation error.

Attaching the following query to the dependency tree produces an additional substrate.

```
SELECT str4[0] STREAM str4 FROM (core1+core0)>2
```

A query attached this way will modify the dependency tree as shown in Fig. 35.

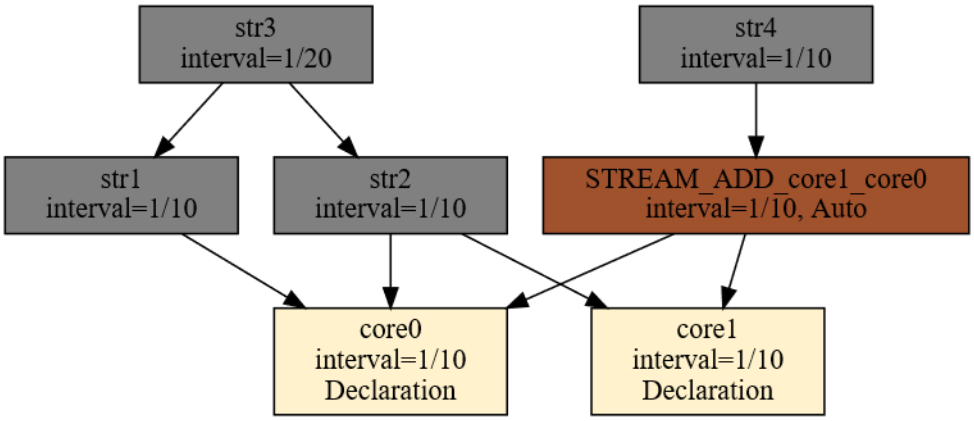



Fig. 35. Dependency with a substrate

The substrate is marked with a different color and an “Auto” label next to the time interval.

The dependency graph must be a directed acyclic graph (DAG). Attempting to define a stream that refers to its own results creates a cycle and results in a compilation error. The detection mechanism is described in the chapter Loop Detection in Compilation.

**Note**

The functionality described here is covered by the test: `subquery`, described in the appendix Integration Tests.

## 5.5. Substrates

I mentioned substrates, ephemerides, and artifacts in the chapter on system architecture. Here I’ll present an example.

First I’d like to draw attention to a certain property of the algebraic expressions introduced. In practice we can write any expression, compile it, and produce a formula for operations on individual elements of the time series that yields the desired result.

In practice, the system carries out only one- or two-argument operations. Examples of one-argument operations are the time shift or the Agse operation — there, the argument is a single data stream. The rest of the operations act on two data streams. During compilation, all algebraic expressions are broken down into ones with two arguments.

The parser accepts both the parenthesized form and unparenthesized chains, e.g. `s1+s2+s3`, `s1#s2#s3`, and `s1+s2+s3+s4`. Such notation is then reduced to a sequence of two-argument operations with automatic intermediate substrates.

The example uses the canonical declarations from the whole chapter — three streams with different types and intervals:

```
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
```

Compilation:

```
$ 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
```

An unannounced stream, `STREAM_HASH_core0_core1`, appeared — this is exactly a substrate. The compiler broke `(core0 # core1) + core2` into two two-argument operations and inserted an intermediate stream. The substrate's delta: $\Delta = (1/10 \cdot 1/5) / (1/10 + 1/5) = 1/15$.

What happens once we attach the query:

```
SELECT merged2[0] STREAM merged2 FROM (core0 # core1) > 2
```

Only one new query gets attached to the plan:

```
merged2(1/15)
        :- PUSH_STREAM(STREAM_HASH_core0_core1)
        :- STREAM_TIMEMOVE(2)
        merged2_0: BYTE
                PUSH_ID(merged2[0])
```

You're probably wondering why only one, and not two again? The answer is optimization. We're reusing the intermediate results from before. This is one of the unexpected benefits of using RetractorDB.

There's one more important thing worth mentioning here. There is a SUBSTRAT directive, whose argument is a string in quotes. You can use the following types: 'memory', 'default', 'direct', 'posix', 'posixshd', 'generic', 'device', 'textsource'. A full description of each type can be found in the chapter Storage Types. The default type, 'default', causes substrates to materialize entirely on disk. That's not the desired behavior in a production system, but it is desired during development and debugging. A useful type is 'memory'. Substrates of this type live only in memory. Their data never lands on disk — everything happens in memory, and there's only as much data as is needed to execute the queries. The remaining types are currently untested and in development.

Adding a query with the same operations but a different name can trigger substrate deduplication. If the program, delta, and schema are equivalent, the compiler redirects the `PUSH_STREAM` references to the existing stream and removes the duplicate.

**Note**

The functionality described here is covered by the tests: `issue96_no_substrat_reduction`, `issue96_substrat_reference`, described in the appendix Integration Tests.

### Substrate reduction

The compiler carries out an optimization called **substrate reduction** (the `deduplicateSubstrats` function). It works as follows: if the user has defined a query structurally identical to a generated substrate, the substrate is removed from the plan and its references are replaced with the user query's name.

### Conditions for reduction

Reduction of a substrate into a user query happens if and only if three conditions are simultaneously satisfied:

1. **The same schema** — the output field types and names are identical.
2. **The same delta** — the streams' sampling rate is the same.
3. **The same processing operations** — the sequence of `PUSH_STREAM` / `STREAM_TIMEMOVE` / `STREAM_HASH`, etc. instructions is identical.

### Reduction example

Consider a query with the canonical declarations:

```
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
```

Without reduction, the compiler would generate three streams: the substrate `STREAM_TIMEMOVE_core0`, `merged`, and `shifted`. The substrate and `shifted` have an identical structure — the same source stream `core0` and the same `>2` operation. After reduction, the substrate is removed, and the reference `PUSH_STREAM(STREAM_TIMEMOVE_core0)` inside `merged` is replaced with `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
```

### An important restriction: only substrates are reduced

Reduction applies exclusively to substrates generated by the compiler (`isSubstrat = true`). Queries explicitly defined by the user are **never** reduced, even if two of them have an identical structure.

Example — two user queries with the same operation:

```
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
```

The compilation result keeps both streams, with no reduction at all:

```
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
```

This semantic decision is deliberate: the user declared two separate output streams, and both are entitled to exist independently in the execution plan.

## Elimination of duplicate substrates

When several queries use the same stream operation — e.g. `core0 + core1` — the substrate-extraction phase (`extractIntermediateStreams`) creates a separate substrate for each of them. Without a subsequent repair phase, the graph would end up with parallel, identical intermediate nodes computing exactly the same value.

### When a substrate is created

A substrate is generated for every query whose program contains more than one stream operator. This applies to the operators: `STREAM_ADD`, `STREAM_SUBTRACT`, `STREAM_HASH`, `STREAM_DEHASH_DIV`, `STREAM_DEHASH_MOD`, `STREAM_TIMEMOVE`, `STREAM_AGSE`. The condition is checked by the `query::isReductionRequired()` function.

The newly created substrate is given a name built from the operation symbol and the operand names, e.g. `STREAM_ADD_core1_core0` (the `composeStreamName` function in `compiler.cpp`). In the parent query's program, the operator token is replaced with a `PUSH_STREAM` token pointing at this substrate.

### The deduplication algorithm

After extracting substrates and determining time intervals, the compiler runs the `deduplicateSubstrats()` step. The algorithm works iteratively — a `while(changed)` loop repeats the search until no more duplicate pairs are found.

On every pass, for each pair of substrates `(it, it2)`, five equivalence conditions are checked in turn:

1. **Time interval** – `it->rInterval == it2->rInterval`
2. **Program length** – the number of tokens in `lProgram` must be identical
3. **Schema length** – the number of fields in `lSchema` must be identical
4. **Program content** – every token is compared by instruction type (`getCommandID()`) and parameter value (`getVT()`)
5. **Schema content** – every field is compared by type (`rtype`), size in bytes (`rlen`), and cardinality (`rarray`)

If all conditions hold, substrate `it` is considered a duplicate of substrate `it2`. The compiler walks through the entire `coreInstance` and, in every `PUSH_STREAM` token referring to the old name (`it->id`), substitutes the new name (`it2->id`). The duplicate is then removed from the query list (`coreInstance.erase(it)`), and the loop starts over.

### Position in the compilation pipeline

Deduplication is the fourth step of an eight-phase pipeline (the `compiler::compile()` function):

```
1. extractIntermediateStreams   – substrate extraction
2. expandSchemaWildcards       – expansion of wildcard symbols in schemas
3. resolveStreamIntervals       – time-interval computation
4. deduplicateSubstrats         – duplicate elimination  ← this step
5. resolveFieldReferences       – field-reference resolution
6. expandIndexWildcards         – expansion of wildcard indices
7. localizeFieldOffsets         – field-offset computation
8. validateConstraints / applyCapacities
```

Deduplication must happen after step 3, because comparing intervals is one of the equivalence criteria — substrates with different intervals are not identical, even if they carry out the same algebraic operation.

### Effect on the dependency graph

Consider the queries:

```
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
```

Both queries require the sum `core0+core1` to be computed first.

The `extractIntermediateStreams` phase creates a separate substrate for each query, producing two identical intermediate nodes in the graph (Fig. 36):

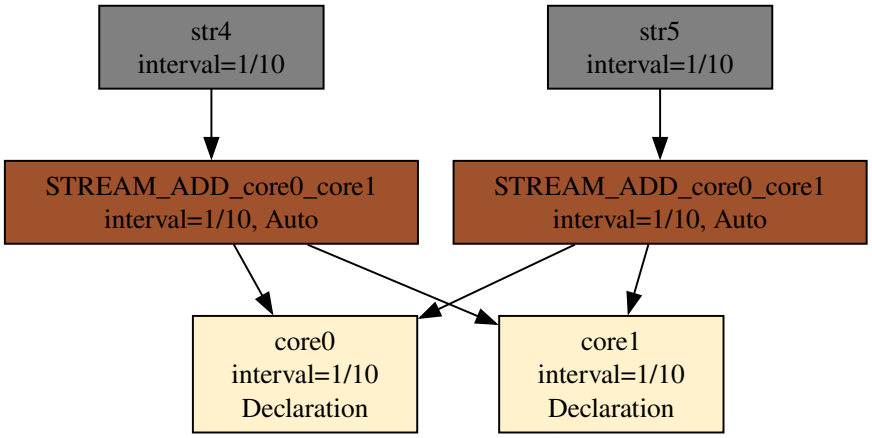



Fig. 36. Graph before deduplication — two identical STREAM_ADD_core0_core1 substrates

Once `deduplicateSubstrats()` runs, one of the duplicates is removed and every `PUSH_STREAM` reference is repointed to the surviving node. A single shared substrate remains in the graph (Fig. 37):

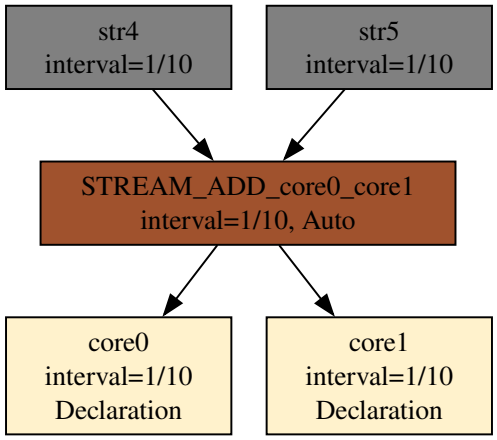



Fig. 37. Graph after deduplication — one shared substrate, generated with: xretractor dedup_after.rql -c -d

The graph after deduplication is exactly what `xretractor -c -d` returns — the compiler always presents the result after all optimization phases.

### Absorption of a substrate by an explicit stream

The inner loop in `deduplicateSubstrats()` does not check the `isSubstrat` flag for candidate `it2` — that check exists only in the outer loop. This means an automatic substrate can be absorbed not only by another substrate, but by **any stream with an identical program and schema** — including a stream explicitly defined by the user.

Consider a query containing only a compound expression:

```
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
```

Here, `extractIntermediateStreams` extracts a substrate `STREAM_ADD_core0_core1` for the expression `core0+core1`. Artifact `str4` depends on it (Fig. 38):

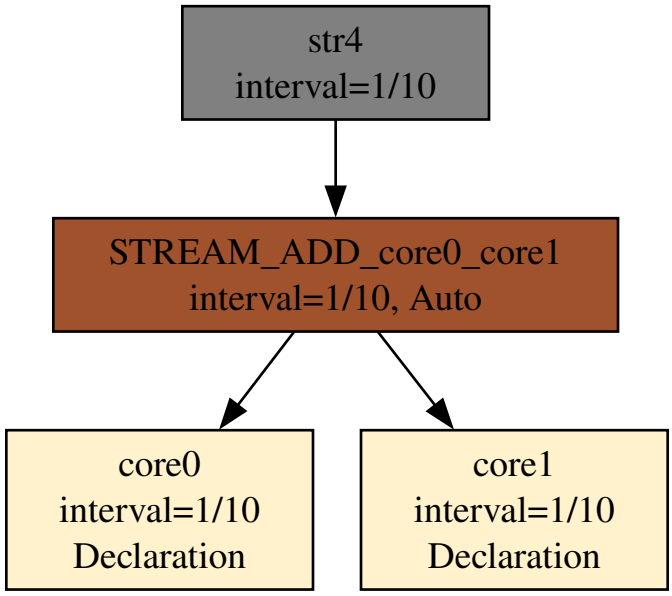



Fig. 38. Graph with the automatic substrate STREAM_ADD_core0_core1

When the user adds an explicit stream declaration that is exactly the same sum:

```
SELECT * STREAM mysum FROM core0+core1
```

the substrate `STREAM_ADD_core0_core1` satisfies every equivalence condition relative to `mysum` — identical interval, identical token program, identical field schema. The `deduplicateSubstrats()` phase removes the substrate and repoints every `PUSH_STREAM` reference to `mysum`. The substrate disappears from the graph entirely (Fig. 39):

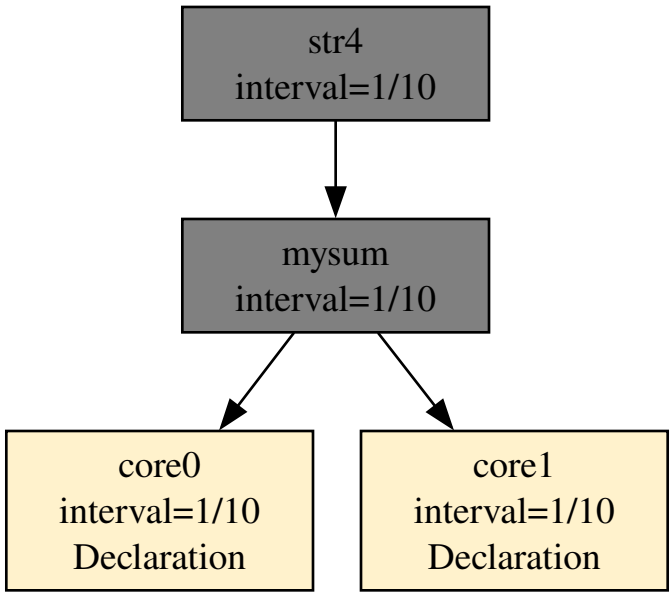



Fig. 39. Graph after adding SELECT * STREAM mysum FROM core0+core1 — the substrate replaced by an explicit stream

A side effect: `mysum` becomes a shared node — it serves both its own consumers and those that previously used the automatic substrate. In exchange, the user gains an explicit name for the intermediate results and can query them via `xqry`.

### Schema update after absorption

Simply repointing the `PUSH_STREAM` tokens is not enough. Every stream stores, in `lSchema`, a sequence of instructions describing how to build the output value of every field — including `PUSH_ID(stream_name, N)` tokens, which say: "take the N-th field from the input buffer named `stream_name`." When a substrate is absorbed, these tokens still refer to the old, removed substrate name. The `localizeFieldOffsets()` step builds an offset map based on the `PUSH_STREAM` tokens in the program — if a `PUSH_ID` key doesn't match any entry in the map, it defaults to offset 0.

#### Error scenario with a non-zero offset

Consider the query:

```
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)
```

The compiler creates a substrate `STREAM_ADD_s1_s2`. Stream `merged` has two sources: `s3` (offset 0) and the substrate `STREAM_ADD_s1_s2` (offset 1, because s3 occupies

position 0). The `buildOutputSchema` function writes the following tokens into `merged.lSchema`:

```
PUSH_ID(STREAM_ADD_s1_s2, 0)   ← field a from the source at offset 1
PUSH_ID(STREAM_ADD_s1_s2, 1)   ← field b from the source at offset 1
```

After absorption, `deduplicateSubstrats()` repoints `PUSH_STREAM` from `STREAM_ADD_s1_s2` to `mysum`. But without updating `lSchema`, the `PUSH_ID` tokens still carry the old name. When `localizeFieldOffsets()` fails to find `STREAM_ADD_s1_s2` in the offset map, it defaults to offset 0 — colliding with `s3`'s fields. Effect: fields `a` and `b` from `mysum` were being read from offset 0 (s3's position) instead of offset 1 (mysum's position).

### The fix: updating lSchema in deduplicateSubstrats

To avoid this discrepancy, after updating the `PUSH_STREAM` tokens, `deduplicateSubstrats()` performs an additional pass over the `lSchema` of every query and rewrites:

- `PUSH_ID(old_name, N)` tokens into `PUSH_ID(new_name, N)` — this covers the fields from `buildOutputSchema` for `STREAM_ADD`,
- `PUSH_ID2("old_name[N]")` tokens into `PUSH_ID2("new_name[N]")` — this covers the symbolic names created by `buildOutputSchema` for `STREAM_TIMEMOVE`, `STREAM_HASH`, `STREAM_SUBTRACT`.

After the fix, the compiler's output for the example above looks correct:

```
merged(1/1)
        :- PUSH_STREAM(mysum)
        :- PUSH_STREAM(s3)
        :- STREAM_ADD
        a: INTEGER
                PUSH_ID(merged[1])
        b: INTEGER
                PUSH_ID(merged[2])
```

Fields `a` and `b` from `mysum` have offset 1 (`merged[1]`, `merged[2]`), which matches `mysum`'s actual position in `merged`'s buffer — after field `c` from stream `s3`.

### Cascaded absorption

**Note**

The functionality described here is covered by the tests: `issue167_dedup_cascaded`, `issue167_dedup_field_names`, `issue167_dedup_nonzero_offset`, `issue167_dedup_positive`, `issue167_triarg`, described in the appendix Integration Tests.

`deduplicateSubstrats()` runs iteratively (`while(changed)`), which allows for multi-step absorption. In the example:

```
SELECT * STREAM mysum   FROM s1+s2
SELECT * STREAM shifted FROM (s1+s2)>1
SELECT * STREAM merged  FROM s3+((s1+s2)>1)
```

in the first round, `mysum` absorbs `STREAM_ADD_s1_s2` and rewrites its names — including in the schema of the intermediate substrate `STREAM_TIMEMOVE_STREAM_ADD_s1_s2`. As a result, in the second round `shifted` can absorb this substrate (the program condition is now satisfied, because both point to `mysum`). After two rounds, no automatic substrate remains in the plan, and `merged` uses `s3` and `shifted` directly.

## 5.6. Asterisk Expansion

Everyone who has written SQL knows the magic * character in that language. Invoking a SELECT command with this argument expands the argument list based on the schemas of the tables produced by relational joins. I wanted to achieve something similar in RQL.

The example uses the canonical declarations used throughout the chapter:

```
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
```

Let’s compile it and see the effect:

```
$ 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
```

The * symbol turned into four fields: `core0_0`, `core0_1`, `core1_2`, `core1_3`. Naming convention: source stream name + absolute position in the output schema. Field types determine the order — `core0` contributes BYTE and INTEGER at positions 0 and 1, `core1` contributes INTEGER and FLOAT at positions 2 and 3. Referring to `merged[2]` in the `result` query gets us a field of type INTEGER — third in order, the first field from `core1`.

**Note**

The functionality described here is covered by the test: `Pattern3`, described in the appendix Integration Tests.

## 5.7. Interval Resolution

Every stream in RetractorDB has an assigned time interval — delta (Δ). The interval determines how often new values are produced. For declared streams (`DECLARE`), the interval is given by the user. For output streams (`SELECT`), the interval is determined by the compiler from the stream-algebra equations.

The examples in this chapter use the canonical declarations from the whole chapter: `core0` (Δ=1/10), `core1` (Δ=1/5), `core2` (Δ=3/10).

### Algorithm

The `resolveStreamIntervals` stage works iteratively:

```
prevUnresolved = ∞
loop:
    unresolvedCount = 0
    sort qTree topologically
    for every query:
        if the source streams' deltas are known:
            determine the output delta from the operator's equation
        otherwise:
            unresolvedCount++
```

```
    if unresolvedCount == 0: done (success)
    if unresolvedCount >= prevUnresolved: error (loop in the graph)
    prevUnresolved = unresolvedCount
```

Every round resolves at least one stream — because the graph is acyclic, and the topological sort guarantees sources are processed before outputs. If the number of unresolved streams stops decreasing, that indicates a cycle — see Loop Detection.

## Operator equations

### Stream sum (+, STREAM_ADD)

```
SELECT ... STREAM c FROM a + b
```

$$\Delta_c = \min(\Delta_a, \Delta_b)$$

The output stream produces values as often as the faster of the input streams.

Example: core0(Δ=1/10) + core1(Δ=1/5) → str1(Δ=1/10)

### Stream synchronization (#, STREAM_HASH)

```
SELECT ... STREAM c FROM a # b
```

$$\Delta_c = \frac{\Delta_a \cdot \Delta_b}{\Delta_a + \Delta_b}$$

The result corresponds to the harmonic mean of the intervals — the stream only produces a value when both inputs are available at the same time.

Example: core0(Δ=1/10) # core1(Δ=1/5) → str1(Δ=1/15)

### Time shift (>n, STREAM_TIMEMOVE)

```
SELECT ... STREAM c FROM a > n
```

$$\Delta_c = \Delta_a$$

A shift does not change the stream's rate — it only shifts the read window by n samples.

### Window aggregates (.max, .min, .avg, .sum)

$$\Delta_c = \Delta_a$$

Aggregates reduce values within a window, but the output stream's interval stays the same as the source's.

#### The AGSE algorithm (`@(step, window)`, STREAM_AGSE)

```
SELECT ... STREAM c FROM a @ (step, window)
```

$$\Delta_c = \frac{\Delta_a \cdot \text{step}}{\text{windowSize}}$$

AGSE (the Episode Series Generation Algorithm) generates sliding windows. The output interval depends on the step and the window size relative to the source.

#### De-hash operators (STREAM_DEHASH_DIV, STREAM_DEHASH_MOD)

The inverse operations of # — they determine what interval one of the input streams had, given the result's interval and the other argument:

$$\Delta_a = \frac{\Delta_c \cdot \Delta_b}{|\Delta_c - \Delta_b|}$$

### Why iteration?

In a query with multiple output streams, one stream may depend on another:

```
DECLARE a INTEGER STREAM core0, 0.1 FILE 'data.dat'
SELECT str1[0] STREAM str1 FROM core0
SELECT str2[0] STREAM str2 FROM str1
```

In the first iteration round, the compiler determines Δ_str1 = 1/10 (because Δ_core0 is known). In the second round — Δ_str2 = 1/10 (because Δ_str1 is now known). Without iteration, str2 would have to be declared before str1, which would limit the language's expressiveness.

## 5.8. Loop Detection in Compilation

The query dependency graph must be a directed acyclic graph (DAG). If a query refers — directly or indirectly — to its own results, a cycle is created. The compiler detects this situation and aborts compilation with an error.

**Note**

The functionality described here is covered by the test: `issue95_loopInCompile`, described in the appendix Integration Tests.

### Example of a loop

```
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
```

The last query defines `broken` as the result of the operation `merged + broken` — the stream depends on itself. The dependency graph contains a cycle (Fig. 40):

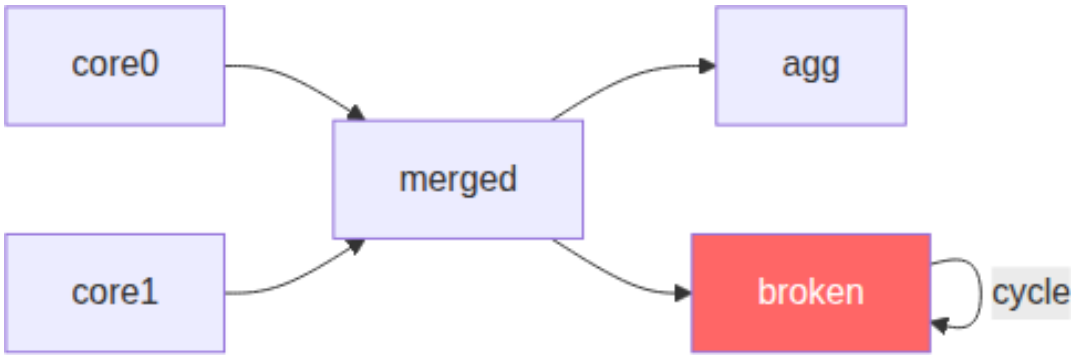



Fig. 40. A cycle in the query dependency graph

### Compilation result

Attempting to compile such a file ends with an error:

```
$ xretractor brokenQuery.rql -c 2>out.txt
$ echo $?
1
$ cat out.txt
[error] Circular dependency: stream interval resolution stalled with 1
>> unresolved streams
```

The message `"Circular dependency in stream definitions"` appears when the `resolveStreamIntervals` stage detects that the number of unresolved streams has stopped decreasing. For how to run compilation and read error messages, see Compilation Debugging.

### Detection mechanism

On every iteration round, the `resolveStreamIntervals` stage counts the streams for which an interval could not yet be determined (`unresolvedCount`). In a valid, acyclic graph, this number decreases every round — at least one stream always gets its delta determined. In a graph with a cycle, streams depend on each other mutually and none can obtain a value — `unresolvedCount` stalls.

```
if (unresolvedCount >= prevUnresolved) {
    SPDLOG_ERROR("Circular dependency: stream interval resolution stalled with
>> {} unresolved streams",
                 unresolvedCount);
    return std::string("Circular dependency in stream definitions");
}
prevUnresolved = unresolvedCount;
```

The >= condition (rather than >) guards against false positives: if the count doesn't decrease by even one, progress is impossible.

### How to fix it

Remove the stream's reference to itself, or to a stream that depends on it. In the example above, the query:

```
SELECT * STREAM broken FROM merged + broken
```

should be replaced with a reference to a stream that exists independently of `broken`:

```
SELECT * STREAM broken FROM merged + core0
```

## 5.9. Aliasing

When we join two data streams with the sum operator, a new data schema appears. We can refer to the successive values of this schema through the name of the data stream, indexed sequentially from the start of the schema.

We can, however, also use the names the stream was built from. A value can be pointed to both by the output stream's name, indexed from the start of the schema, and by the source stream's name, shifted relative to its join position.

The example uses the canonical declarations used throughout the chapter:

```
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
```

After compilation we get:

```
$ 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]` and `core0[0]` both end up as `PUSH_ID(merged[0])` — they are the same field. But `core1[0]` — the first field of `core1`'s schema — ends up as `PUSH_ID(merged[2])`, not `merged[0]`. The compiler translated the local index `core1[0]` into an absolute position in the combined schema: `core0` occupies positions 0 and 1, so `core1` starts at position 2. What if we replace the + operator with #? I encourage you to experiment.

**Note**

The functionality described here is covered by the test: `Pattern7`, described in the appendix Integration Tests.

## 5.10. Underscore Symbol Processing

In some queries you can use the underscore symbol. This technique is syntactic sugar. Similarly to expanding the * symbol, a single reference results, after compilation, in many fields appearing. How many of these fields are produced depends on what was joined with what, and in what order, in the FROM clause.

The example uses the canonical declarations used throughout the chapter — `core0` has two fields (BYTE, INTEGER), `core1` has two fields (INTEGER, FLOAT), the schemas have equal cardinality:

```
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
```

After compiling:

```
$ 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
```

The _ symbol expanded into two fields: `scaled[0] * scaled[2]` (i.e. `a * c`) and `scaled[1] * scaled[3]` (i.e. `b * d`). References to `core0` and `core1` were translated, via aliasing, into absolute positions in the combined schema. The resulting types are INTEGER (`BYTE * INTEGER`) and FLOAT (`INTEGER * FLOAT`) — the result of type promotion, described in a separate subchapter.

Once the _ operator appears in a formula's array index, the compiler repeats the formula for every field of the arguments. The schemas of both arguments must have equal cardinality — that is, core0 and core1 must have schemas of the same size, and the types will be promoted to the highest one. I'll cover type promotion shortly.

This functionality is mainly used when building queries that implement signal-filter algorithms. There, a series of mathematical operations comes into play. The functionality behind underscore-symbol processing is not required to achieve RetractorDB's full functionality. However, it significantly simplifies building specific queries where operations on two schemas need to be combined. An example use case will be presented when signal-processing algorithms are introduced.

## 5.11. Type Promotion

What happens when we multiply data of type BYTE with data of type INTEGER? RetractorDB follows strict type-promotion rules. Multiplying a BYTE-type field by a value of a field that is of type INTEGER produces a schema field of type INTEGER. This happens at compile time.

At present, RetractorDB supports the following data types:

| Type | Description |
|---|---|
| BYTE | values 0–255 |
| INTEGER | 4-byte values for signed numbers |
| UINT | like INTEGER, for unsigned numbers |
| RATIONAL | rational numbers |
| FLOAT | floating-point numbers |
| DOUBLE | double-precision floating-point numbers |
| STRING | character strings |

The STRING and RATIONAL types still need review, fixes, and test coverage. During development I focused my effort on number processing. In the future I want to add complex numbers and rational Eisenstein complex numbers to this set as well.

An example of type promotion in practice — the `scaled` query from the chapter Underscore Symbol Processing:

```
SELECT core0[_] * core1[_] \
STREAM scaled \
FROM core0 + core1
```

`core0` has fields BYTE and INTEGER, `core1` has fields INTEGER and FLOAT. After expanding `_`, the compiler determines the output fields' types:

| Expression | Left type | Right type | Result type |
|---|---|---|---|
| `scaled[0] * scaled[2]` | BYTE | INTEGER | INTEGER |
| `scaled[1] * scaled[3]` | INTEGER | FLOAT | FLOAT |

## 5.12. Compilation Debugging

The compiler transforms an `.rql` file into an execution plan through several stages. The effect of each stage is visible through `xretractor`'s diagnostic flags. The tools described here let you answer questions like: why does the schema look different from what I wrote? where does this delta come from? why did a substrate appear?

### The basic tool: the `-c` flag

The `-c` (`--onlycompile`) flag stops `xretractor` after compilation and prints the compiled plan to standard output — without starting processing:

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

An exit code of `0` means success. Code `1` means a compilation error. Error messages go to stderr:

```
xretractor -c query.rql 2>errors.txt
echo $?
```

Compilation can be invoked even while another `xretractor` process is already running — the `-c` flag does not attempt to acquire the execution lock.

### How to read the compilation plan

For the canonical `query.rql` from this chapter, the plan looks as follows:

```
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
```

Every block has a fixed format:

```
streamName(delta)
        :- streamOperation(arg)
        outputFieldName: TYPE
                instruction
                ...
```

| Element | Meaning |
|---|---|
| `streamName(delta)` | The stream's name and its interval as a fraction: `1/10` = 0.1 s = 10 Hz |
| `:- PUSH_STREAM(x)` | Pushes stream `x` onto the stream stack; appears once per FROM argument |
| `:- STREAM_ADD` | The stream-sum operator (+ in FROM) |
| `:- STREAM_HASH` | The stream-synchronization operator (# in FROM) |
| `:- STREAM_TIMEMOVE(n)` | Time shift (>n in FROM) |
| `field: TYPE` | An output-schema field, after type promotion |
| `PUSH_ID(s[n])` | Pushes the value of field `n` from stream `s` onto the stack — the effect of aliasing is visible here |
| `PUSH_VAL(x)` | Pushes the constant `x` onto the stack |
| `ADD`, `MULTIPLY`, … | An arithmetic operation: pops two arguments off the stack, pushes the result |

Ephemeris blocks (`DECLARE`) appear at the end of the plan — they contain the field list and the path to the data file.

**Aliasing in the plan**: if two output fields point to the same `PUSH_ID`, they are aliases. In the example, `result_0` and `result_2` are both `PUSH_ID(merged[0])` — confirmation that `merged[0]` and `core0[0]` are the same position. See Aliasing.

**Substrates in the plan**: an automatically generated substrate appears as a block with a name like `STREAM_HASH_core0_core1` — with no corresponding `SELECT` in the source file. See Substrates.

### Visualizing the dependency graph

Instead of text, you can generate a graph in DOT format and process it with `graphviz`:

```
xretractor -c -d -f -s query.rql > out.dot && dot -Tsvg out.dot -o out.svg
```

Available flags that modify the DOT output:

| Flag | Full name | Meaning |
|---|---|---|
| `-d` | `--dot` | generate DOT output instead of a text plan |
| `-f` | `--fields` | show stream fields in the graph nodes |
| `-s` | `--streamprogs` | show stack-instruction sequences in the nodes |
| `-u` | `--rules` | show RULE rules |
| `-p` | `--transparent` | transparent background — for embedding in documents |

The graph shows dependencies between streams as edges directed from sources to results. Substrates have a different color than streams explicitly defined by the user. See Dependency Tree Construction.

**Note**

The functionality described here is covered by the test: `issue31_doc`, described in the appendix Integration Tests.

## Verifying intervals

If an output stream's delta is unexpected:

1. Check the source streams' deltas — visible in the DECLARE blocks at the end of the plan.
2. Check the operator in the FROM clause — every operator has a different delta equation.

Example: `core0(1/10) # core1(1/5)` gives a delta of `1/15` (the harmonic mean), not `1/10`. If you expected `1/10`, use `+` instead of `#`. Full equations — see Interval Resolution.

## Common compilation errors

### A cycle in the dependency graph

```
[error] Circular dependency: stream interval resolution stalled with N
>> unresolved streams
```

A stream refers, directly or indirectly, to itself. Generate the graph via `-d` — the cycle will be visible as a loop. See Loop Detection.

### Unknown stream

A reference to a stream that hasn't been declared yet. `.rql` files are processed sequentially — a `SELECT` cannot refer to a stream defined further down in the file. Move the `DECLARE` or `SELECT` earlier.

### Schema cardinality mismatch with _

Both streams in the expression `core0[_] * core1[_]` must have schemas of the same cardinality. Check how many fields each argument has in the plan's DECLARE blocks. See Underscore Symbol Processing.

### Data file unavailable

This error **does not appear with `-c`** — the flag verifies the query's correctness, it does not check whether the data files exist. The file-access error only appears when processing is started without `-c`.

# Chapter 6

# Query Execution

The execution process is based on a continuous traversal of the query tree, and a sequential, hierarchical invocation of the procedures that build successive stream tuples and successive data schemas.

The description of the algorithm needs to start with the sequencing procedure. In a system executing queries with a variety of values defining the time period between successive created and arriving data, a way to determine successive intervals is needed.

Let's start by analyzing the following example. Suppose the system has two data streams. One arrives every second, the other every two seconds. The sequencing algorithm should propose a one-second time interval between invocations of the procedure intended for the first stream, and a two-second interval for the second stream. In practice, a one-second time grid will emerge — in which every one-second node is filled with the tuple-building procedure for the first stream, and, on that same time grid, the second stream attaches its procedures every second node.

The time grid determined during compilation is very important — it defines how often, and at what intervals, data streams will be processed, and which nodes will be covered by the generated stream-processing procedures.

Let's consider a more complex example. Suppose there are three data streams. The first arrives at a rate of ⅓, the second at ½, and the third at ⅔. Determining the grid and placing successive processing procedures requires a more elaborate solution. The value ⅔ can be simplified to ⅓, since there's a natural divisor between these values. There is, however, no natural divisor between ½ and ⅓. The grid value that gets determined is ⅕. This is the largest possible rational number that, if a grid is built on the rational-number axis, accommodates regular time series with arrival rates of ½ and ⅓.

All streams are overlaid onto the determined grid, and the set of minimal time gaps is determined for the queries with natural multipliers. In our case, the minimal set of time gaps for queries with multipliers (⅓, ½, ⅔) is the set (⅓, ½). The gaps ⅓ and ⅔ will share a time slot on the grid.

Analyzing the discussion below may make clearer the comments, mentioned earlier in the chapter, generated for the swirly program showing the generated marble dia-

grams.

## 6.1. Query Tree Traversal Algorithm

### General overview

The query-tree traversal algorithm is carried out by two cooperating components: `dataModel` (processing logic) and `executorsm` (the time loop and IPC). Before entering the main loop, the system performs a **zero step**, after which it iterates cyclically over the minimal set of time intervals (Fig. 41).

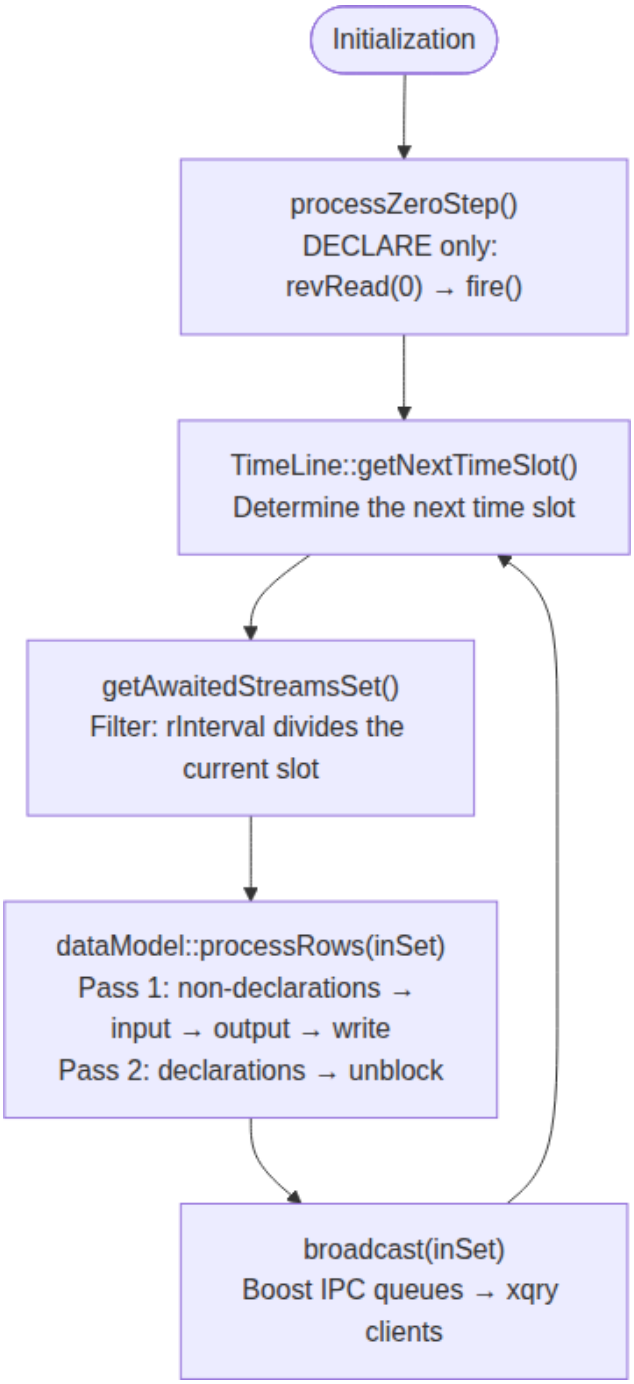



Fig. 41. The query tree traversal algorithm – general overview

---

### Data structure: qTree

`qTree` (`src/retractor/lib/qTree.cpp`) extends `std::vector<query>` and is a **vector of topologically sorted queries**. Sorting is done via DFS over the dependency graph built from `query.getDepStream()` (Fig. 42).

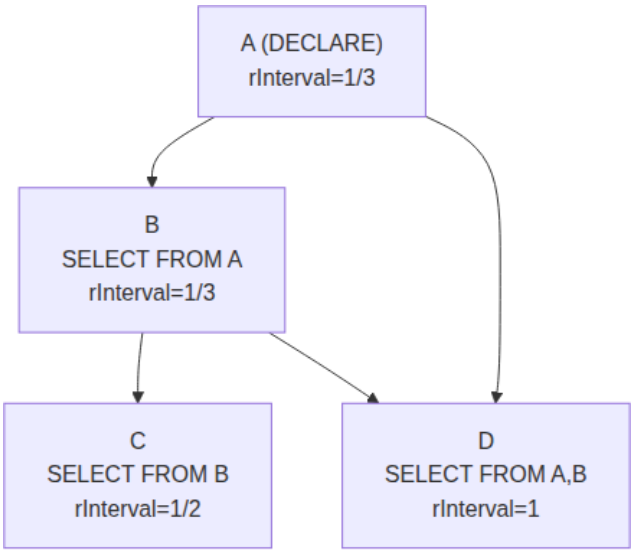



Fig. 42. Example dependency graph for qTree

After the topological sort, the order in the vector is: `[A, B, C, D]`. Query C, which depends on B, always ends up after B in iteration — this guarantees correctness of the computations.

The `getAvailableTimeIntervals()` method extracts the unique `rInterval` values from all queries (excluding compiler directives and zero values) — the result is the input to the `TimeLine` constructor.

---

### The minimal time grid: TimeLine / CRSMath

`TimeLine` (`src/retractor/lib/CRSMath.cpp`) manages rational time intervals. The constructor reduces the set of intervals — removing multiples and keeping only the coprime ones:

```
Input: {1/2, 1, 4}  →  Output: {1/2}
(1 = 2 × 1/2, so redundant; 4 = 8 × 1/2, so redundant)

Input: {1/2, 1/3}  →  Output: {1/2, 1/3}
(neither is a multiple of the other)
```

`getNextTimeSlot()` determines the next slot as `min(delta × counter[delta])` over all deltas. The diagram below illustrates the slots for deltas `{1/2, 1/3}` and the active queries in each of them (Fig. 43):

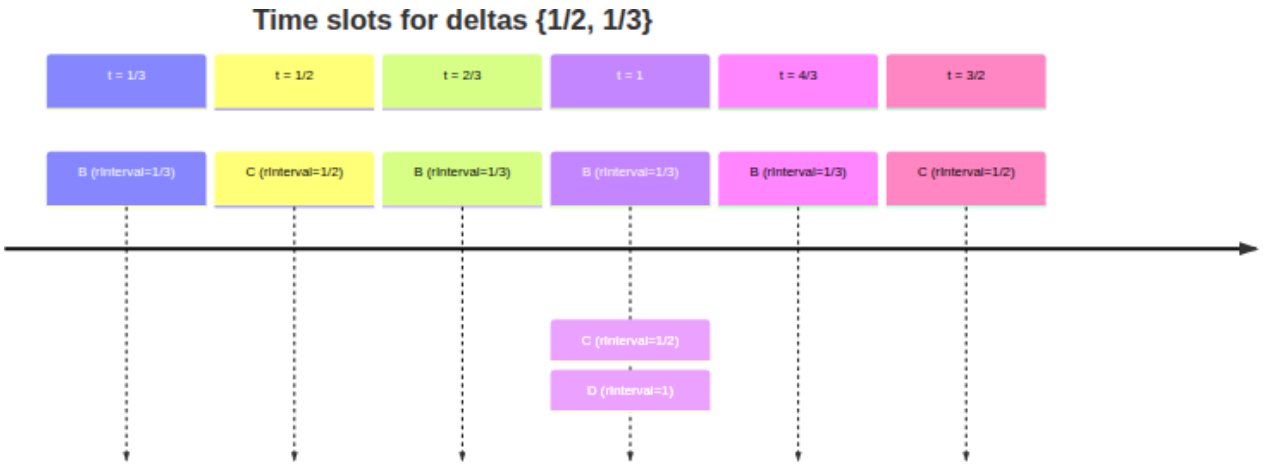



Fig. 43. The minimal time grid for deltas {1/2, 1/3}

The check `isThisDeltaAwaitCurrentTimeSlot(inDelta)` returns `true` when `ctSlot_ / inDelta` has a denominator equal to 1 (the slot is an integer multiple of the query's delta).

---

### The zero step: `processZeroStep()`

Before entering the `executorsm::run()` loop, `processZeroStep()` is called (`dataModel.cpp`, line ~85). It processes **declarations only** (`DECLARE` input streams):

```
for (auto &q : coreInstance_) {
    if (!q.isDeclaration()) continue;
    qSet[q.id]->bufferState = flux;   // unblock physical read
    qSet[q.id]->revRead(0);           // read from index 0
    qSet[q.id]->fire();               // copy chamber_ → outputPayload
    assert(qSet[q.id]->bufferState == armed);
}
```

After this step, every declaration has `bufferState = armed` — the data from the physical source is in `outputPayload`.

---

### The main loop: filtering and processing

#### Query filtering: `getAwaitedStreamsSet()`

For the current slot `tl` (`executorsm.cpp`, line ~88):

```
std::set<std::string> retVal;
for (auto &q : *coreInstancePtr)
    if (TimeLine::isThisDeltaAwaitCurrentTimeSlot(q.rInterval))
        retVal.insert(q.id);
```

```
return retVal;
```

The result `inSet` is the set of query identifiers active in this slot — a subset of all queries.

### Processing: `processRows(inSet)`

The function performs **two passes** over `inSet` (`dataModel.cpp`, line ~98), shown in Fig. 44:

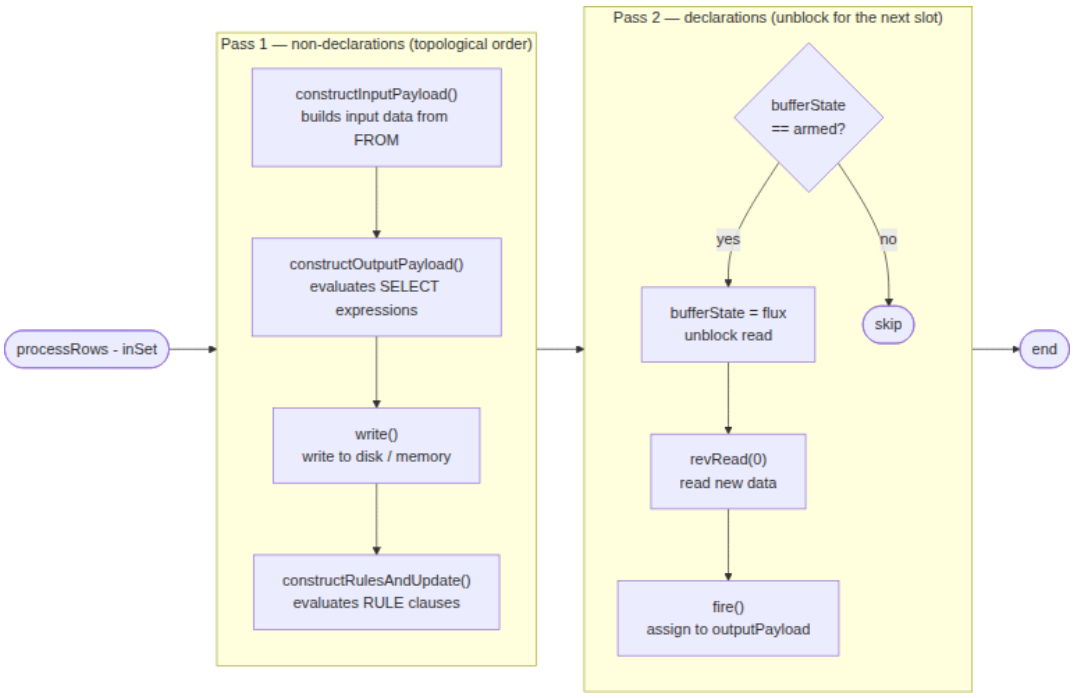



Fig. 44. The processRows algorithm – two processing passes

Declarations are only unblocked once every dependent query has consumed their `outputPayload` in pass 1.

---

## Broadcasting results: `broadcast()`

After every `processRows()`, `broadcast(inSet)` is called (`executorsm.cpp`, line ~449) — the algorithm is shown in Fig. 45:

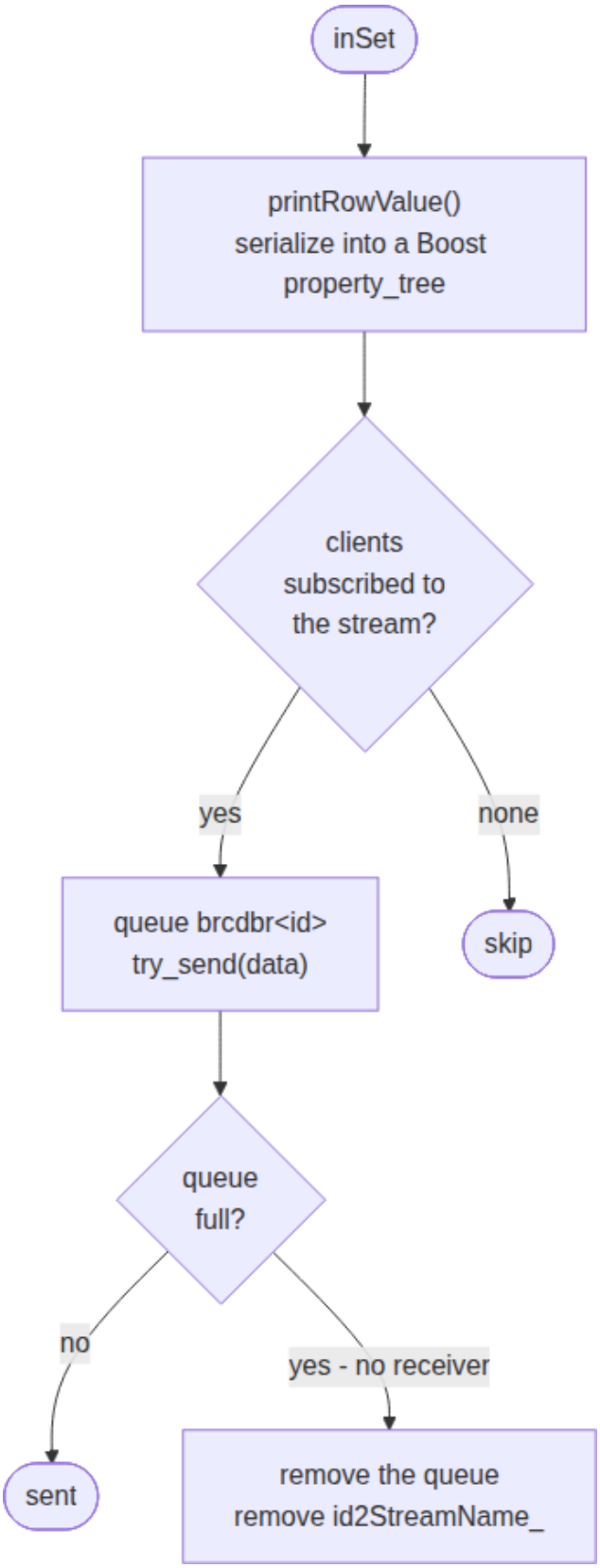



Fig. 45. The broadcast algorithm – distributing results via Boost IPC

`printRowValue()` builds a structure with the stream name, field count, values, and a null bitmap, serializes it in Boost info format, and sends it via a `boost::interprocess::message_queue`.

---

### Full example: queries A, B, C, D for deltas {1/2, 1/3}

Fig. 46 shows the complete call sequence for four queries A, B, C, D laid out on a time grid with deltas {1/2, 1/3}.

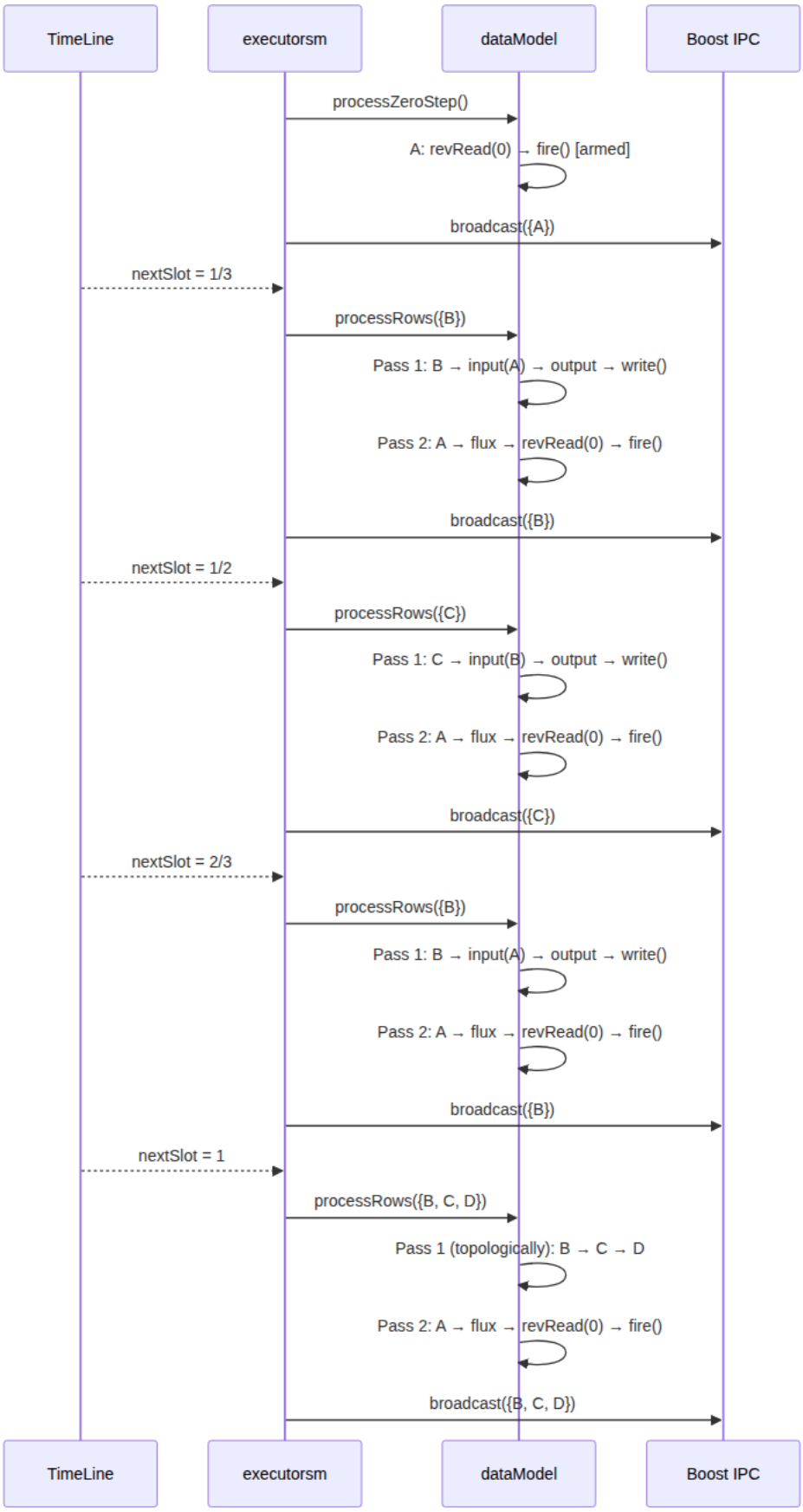



Fig. 46. Full execution example for queries A, B, C, D with deltas {1/2, 1/3}

The dependency tree determines the order of pass 1. Time intervals from the Beatty algebra determine which nodes of the tree are active in a given slot.

---

## Algebraic realization — tying the code to the equations

Every key part of the algorithm described on this page is a direct realization of equations from the algebra of regular time series and the formal proofs.

### Algebraic operators in `SOperations.hpp`

The file `src/include/SOperations.hpp` encodes the algebra operators directly as functions on rational numbers:

| Operator | Symbol | Function in code |
|---|---|---|
| Interleaving | φ | `Hash(Δa, Δb, i, retPos)` |
| Left-hand de-interleaving | Θ | `Div(Δa, Δb, i)` |
| Right-hand de-interleaving | ~Θ | `Mod(Δa, Δb, i)` |
| Difference | δ | `Subtract(Δa, Δb, i)` |
| Aggregation and serialization | Ψ | `agse(offset, step)` |

Each of these functions is a literal translation of the formula from the algebra. `Div` implements left-hand de-interleaving:

```
return i + ceilR((i + 1) * deltaA / deltaB);
```

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

`Mod` implements right-hand de-interleaving:

```
return i + floorR(i * deltaB / deltaA);
```

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

`Hash` implements the test from the definition of interleaving — the condition $\lfloor iz \rfloor = \lfloor (i+1)z \rfloor$ with $z = \Delta_b/(\Delta_a+\Delta_b)$ — and returns the corresponding offset into stream A or B.

The helper functions `floorR()` and `ceilR()` operate exclusively on `boost::rational<int>`, never passing through `double`. This is a direct realization of the requirement from Theorem 2: an implicit cast to `float` breaks the assumptions of Fraenkel's theorem — materialization into floating-point form must be deferred until the floor or ceiling operation is explicitly applied.

### `TimeLine` as the minimal basis of a covering system

The `TimeLine` constructor determines the **primitive set of intervals** — removing every delta that is an integer multiple of another delta in the set. An interval is primitive when no smaller interval in the set divides it with a natural quotient. This is the computation of the minimal covering system in the sense of Fraenkel's theorem: only primitive deltas generate independent Beatty sequences, and only they are needed to determine the complete time grid.

The `getNextTimeSlot()` method — marked with the comment `// MAGIC Warning` in the source — generates successive grid points as:

$$t_k = \min_{\delta \in \mathrm{sr}} \left( \delta \cdot \mathrm{counter}[\delta] \right)$$

where `sr` is the primitive set of intervals, and $\mathrm{counter}[\delta]$ counts the "hits" recorded so far for each delta. The two-phase loop — first determining the minimum, then incrementing the counters separately — guarantees correct handling of collisions: several deltas can determine the same slot at once.

**Info**

The `// MAGIC Warning` comment in `CRSMath.cpp`'s source means the algorithm is correct for a non-obvious reason. Intuition alone is not enough — correctness is guaranteed by Fraenkel's theorem. Because `sr` contains only primitive intervals (none a multiple of another), the counters for the individual deltas never "get ahead of each other" in a way that would skip or duplicate a slot. A collision — when two deltas point to the same slot — is a legitimate case, handled by the second loop. The "magic" is that the simple formula `min(δ·counter[δ])`, with automatic incrementing, is equivalent to a full Beatty-sequence generator for the entire covering system.

### `isThisDeltaAwaitCurrentTimeSlot()` as a Beatty-sequence membership test

```
boost::rational<int> value = ctSlot_ / inDelta;
return (value.denominator() == 1);
```

The test checks whether $t_{\mathrm{slot}}/\Delta \in \mathbb{N}$ — whether the current slot is an integer multiple of the query's delta. In the language of Beatty-sequence theory: a point $t$ belongs to the sequence of density $\Delta$ if and only if $t/\Delta$ is a natural number. The condition on the denominator equaling 1 follows from `boost::rational` arithmetic — the fraction is always in reduced form, so a denominator of 1 means exactly an integer, with no rounding involved.

## 6.2. Ad Hoc Queries

By Ad Hoc queries we mean queries directed at a running system. In the typical scenario originally envisioned during system development, the initial working assumption

was that a system user would know all the queries and data sources needed to obtain the processed time series.

During development, however, additional scenarios emerged, assuming that the system's operation should not be interrupted, and that additional queries should be attached to the query execution plan. We call this kind of functionality Ad Hoc queries — attached to the system while it is running, without interrupting its operation.

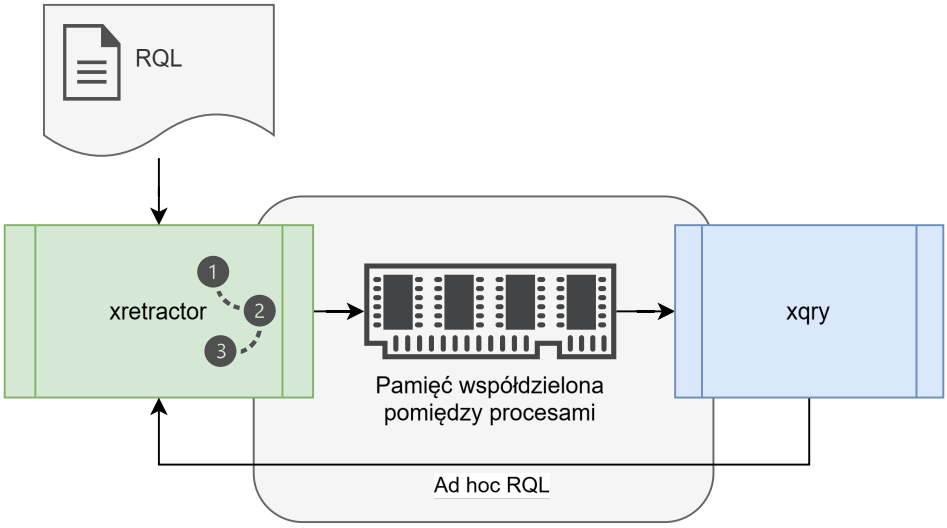



Fig. 47. Control flow for Ad Hoc queries

Fig. 47 shows the control flow described above. A file with queries and directives is first directed to the xretractor process. Then, through shared memory, the xqry process pulls data from xretractor. Using that same process, we can send a command to the xretractor process. In this command we include the text of the additional query that xretractor should attach to the tree being processed.

### Example

We'll start the example by preparing a simple query:

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

We'll save the query file under the name qplan1.rql. For the query to run correctly, we also need to prepare the files data1.txt and data2.txt. I suggest filling data1.txt with consecutive numbers from 1 to 6, each on a new line, and filling data2.txt with numbers from 10 to 15. In a directory prepared this way, we run the command:

```
$ xretractor qplan1.rql
```

If we previously performed some operations in this directory and created a str1 stream with a different schema, we'll get an error titled "Error in data descriptor file". It will

also show information about the differences between the two descriptors. In that case, the files str1 and str1.desc should be deleted and the command run again.

The xretractor process will begin processing data. At this point, open another terminal and issue the command:

```
$ xqry -d
|str1|1|48|24|         |0|
|   A|1|-1| 3|data1.txt|1|
|   B|2|-1| 2|data2.txt|1|
ok.
```

This will display, in tabular form, what's currently being processed in the system — how many bytes have already arrived, which files the data is being read from, and how much data has already been processed. If a more descriptive format is desired, we can issue the following command:

```
$ 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
```

The response is given in YAML form.

To add another query to the system, we need to issue the command:

```
$ xqry -a "SELECT * STREAM str2 FROM A#B"
snd: adhoc SELECT * STREAM str2 FROM A#B
rcv: db OK
```

A command in this form sends a new query to the xretractor process. Upon receiving it, the system compiles it and merges it into the query plan tree.

If we check the system's state again, we'll see the following picture:

```
$ 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.
```

Or like this:

```
$ 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
```

Taking a closer look at the queries via the xqry command, we'll see the following system response for the str1 query:

```
$ 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
```

and for the str2 query:

```
$ xqry -t str2
---
apiVersion: xqry/v1
stream:
  name: str2
```

```
  delta: 2/3
query: SELECT * STREAM str2 FROM A#B
fields:
  str2.a:
    type: BYTE
```

As you can see, the additional query str2 was correctly merged into the existing query execution plan. You can also see that far less data has accumulated compared to str1.

**Note**

The functionality described here is covered by the test: `issue6_adhoc`, described in the appendix Integration Tests.

## 6.3. Alerting Implementation

The alerting mechanism (the `RULE` directive) is an integral part of the main processing loop. It is not a separate background process — rules are evaluated **synchronously**, in the same time-grid iteration as the `SELECT` computations. This guarantees that an alert always refers to data that was just computed, not to the previous cycle.

---

### Where RULE sits in the processing cycle

Recall the `processRows()` function outlined in the chapter Query Tree Traversal Algorithm. For every non-declaration query, four steps are carried out in sequence (Fig. 48):


Fig. 48. The order of processing steps for a single query

The fourth step — `constructRulesAndUpdate()` — is exactly where all rules attached to the current query are executed. It is called after the `SELECT` results have been written to disk, which means a rule always evaluates against a **complete, just-computed sample** of the stream.

---

### Evaluating the WHEN condition

Every rule contains a list of tokens describing a logical expression (the `condition` field of the `rule` struct). At the moment of evaluation, the system:

1. Fetches the current query's `outputPayload` — the current sample of the stream.
2. Passes the condition to the `expressionEvaluator::eval()` engine — **the same engine** that computes `SELECT` expressions.

3. Casts the result to a boolean (`boolCast`): any non-zero numeric value is `true`, zero is `false`.

If the condition is satisfied, the action associated with the rule is executed (`DO SYSTEM` or `DO DUMP`). If not, the rule is skipped with no side effects at all. The full flow is shown in Fig. 49.

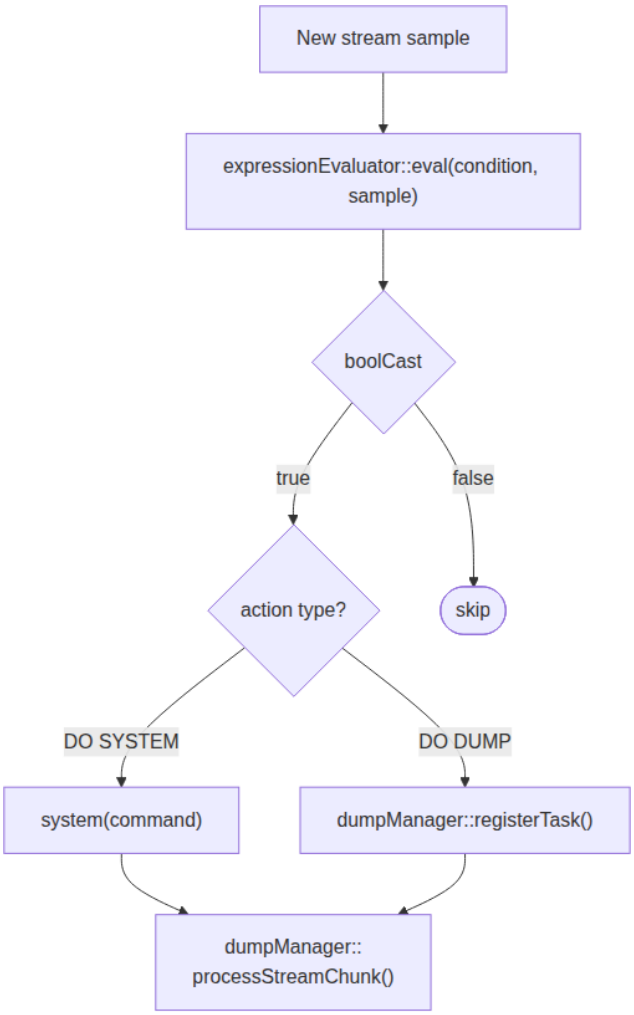



Fig. 49. Rule evaluation flow

---

## The DO SYSTEM action

The `DO SYSTEM` invocation is the simplest: the system calls `::system(command)` directly on the processing thread. The call is **synchronous** — xretractor waits for the process to finish before moving on to the next rule.

The command's exit code is checked: - `0` — success, no log entry. - `≠ 0` — xretractor logs an error via spdlog with the exit code. - A `system()` failure (e.g. no shell available) — logged as a critical error.

**Warning**

The command is executed synchronously. Long-running scripts (e.g. sending large files, network calls with a timeout) will delay the entire processing cycle. In such cases, it is recommended to launch the process in the background: `DO SYSTEM 'my_script &'`.

---

## The DO DUMP action — detailed algorithm

`DO DUMP` is more complex, since it requires gathering data **from the past** (moments before the event) and **from the future** (moments after the event). This is handled by the `dumpManager` class.

### Phase 1: historical data (when the task is registered)

At the moment the rule fires — right after the condition is found to be true — `dumpManager::registerTask()`:

1. Creates the destination file on disk (POSIX `open()` with the `O_CREAT | O_TRUNC` flags).
2. If `step_back < 0`, reads `|step_back|` samples from the stream's historical buffer. Historical data exists because every stream keeps a window of previous samples needed for AGSE window computations.
3. Writes the historical samples to the file **from oldest to newest** (i.e. from `step_back` to −1).
4. Computes how many future samples still need to be collected (`dumpedRecordsToGo = |step_forward - step_back| - |step_back|`).
5. If `step_back ≥ 0` (a delayed start), it sets `delayDumpRecordsToGo = step_back`.

```
Example: DUMP -3 TO 2
  At registration: write samples t-3, t-2, t-1  (history)
  Still to collect from the future: 2 samples (t, t+1)
  dumpedRecordsToGo = 2
```

### Phase 2: future data (subsequent loop iterations)

After registration, the task goes into the `bookOfTasks[streamName]` queue. On every subsequent iteration of the time grid (when the stream produces a new sample), `dumpManager::processStreamChunk()` is called:

1. For every active task in the queue (`dumpedRecordsToGo > 0`):
   - If `delayDumpRecordsToGo > 0` — decrement and skip (start delay).
   - Otherwise — write the current sample to the file and decrement `dumpedRecordsToGo`.
2. Once `dumpedRecordsToGo` reaches 0 — close the file descriptor and remove the task from the queue.

The full sequence for `DUMP -3 TO 2` is shown in Fig. 50.

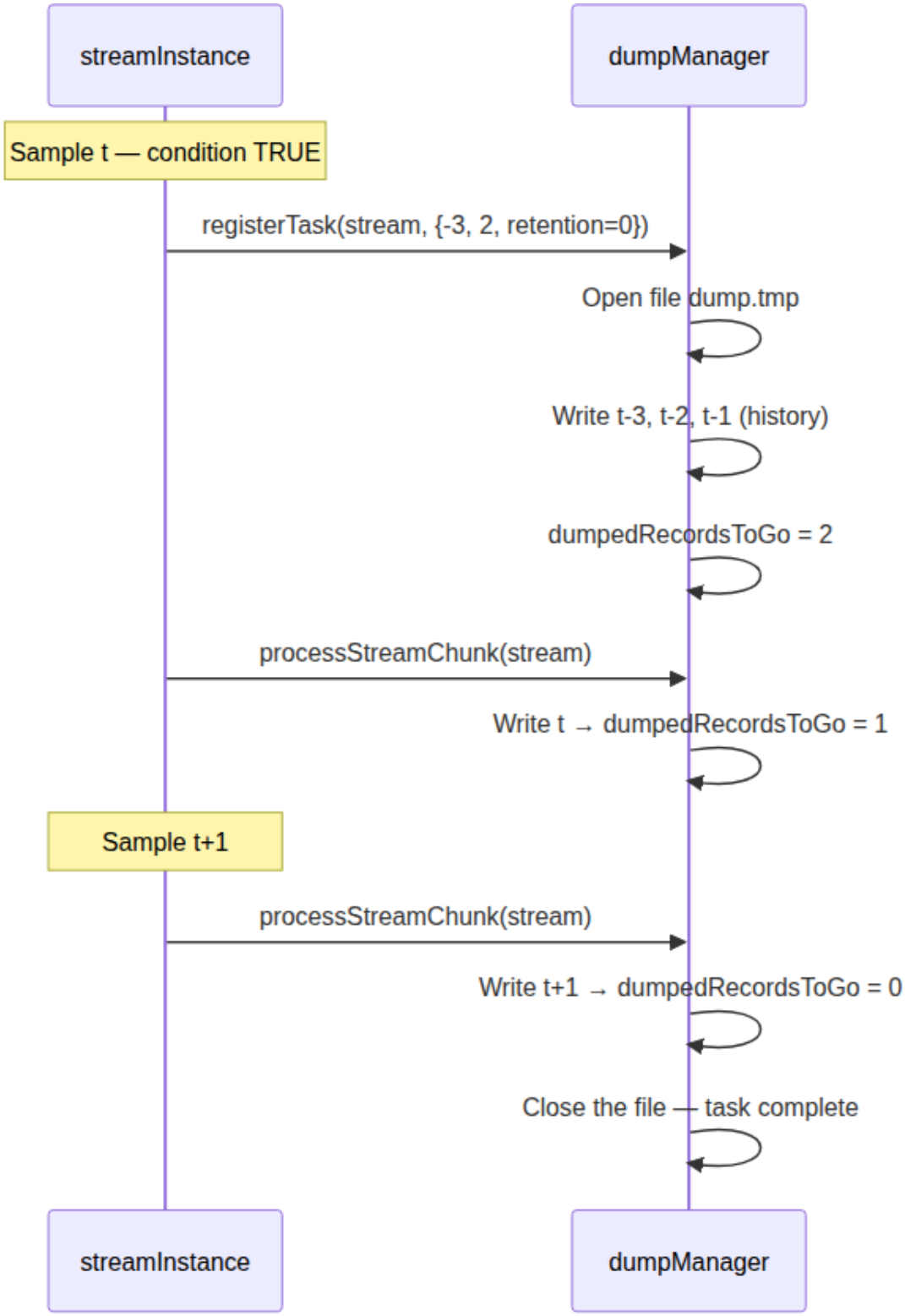



Fig. 50. Data-collection sequence for DO DUMP –3 TO 2

### The delayed-start case (step_back ≥ 0)

When `step_back` is non-negative, the dump does not start at the moment of the event, but `step_back` samples **after** it:

```
Example: DUMP 2 TO 5
  At registration: delayDumpRecordsToGo = 2
  Sample t   → skip (delay=2→1)
  Sample t+1 → skip (delay=1→0)
  Sample t+2 → write (dumpedRecordsToGo = 3→2)
  Sample t+3 → write (dumpedRecordsToGo = 2→1)
  Sample t+4 → write (dumpedRecordsToGo = 1→0) — done
```

---

## Retention (RETENTION N)

Without a `RETENTION` clause, every trigger of a rule overwrites a single file `<stream>_<rule>_dump.tmp`. The `bookOfTasks` queue's capacity is then 1 — a new task evicts the old one (and closes its descriptor).

With a `RETENTION N` clause: - The `bookOfTasks` queue's capacity is set to `N`. - The file number rotates modulo `N`: `_dump_0.tmp`, `_dump_1.tmp`, …, `_dump_(N-1).tmp`. - When the `N`-th task enters the queue, the oldest (still-unfinished) one is **removed** — the `dumpTask` destructor closes its open descriptor.

This means that, with frequent events and a small `N`, an unfinished dump can get interrupted. `N` should be chosen so that the time to collect a single dump (`|step_back| + step_forward` cycles) is shorter than the interval between events multiplied by `N`.

---

## Dump file format

The file contains raw binary records with no header at all — every record has the size determined by the descriptor (`descriptor.getSizeInBytes()`). The format is identical to the format used by stream artifacts, which lets you read it with the `xtrdb` tool after manually specifying the schema:

```
$ xtrdb
> storage <path>
> open <stream>_<rule>_dump { <type> <field> }
> list
> quit
```

---

## Multiple rules — evaluation order

Multiple rules can be attached to a single stream. All of them are evaluated in a single `constructRulesAndUpdate()` iteration, in the order they were declared in the `.rql` file.

Every rule is independent — one being satisfied does not affect the evaluation of the others (Fig. 51).

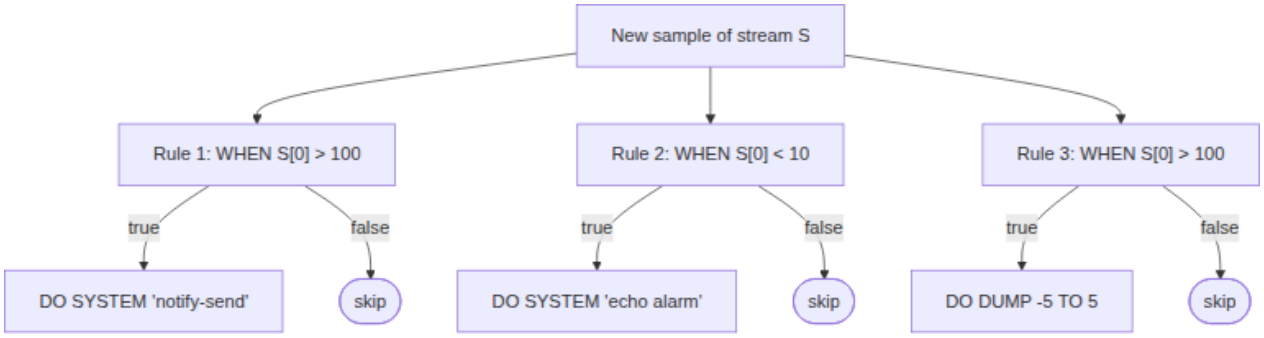



Fig. 51. Independent evaluation of multiple rules on the same stream

---

**Practical limitations and notes**

| Situation | Behavior |
|---|---|
| Condition satisfied twice in a row (e.g. a measurement staying above the threshold) | Every sample registers a new DUMP task — files overlap when RETENTION is absent |
| A `DECLARE` input stream used as an `ON` target | Compilation error — rules can only be attached to `SELECT` streams |
| Insufficient history (buffer shorter than `\|step_back\|`) | The dump contains as many samples as are available; no error |
| Destination file unavailable (missing STORAGE directory) | Critical `FatalError` — xretractor exits |
| DO SYSTEM returns a non-zero code | Error logged via spdlog; processing continues |

## 6.4. AGSE Sliding Data Window

A sliding data window is a concept widely used in systems that process streams or time series. The idea is to group data into time windows, giving the user the ability to process it in frozen snapshots.

RetractorDB supports this data-processing model through the **AgSe** operator (Aggregation and Serialization). This operator is two-argument and operates on a stream. Denoted with the `@` symbol, it has the form:

```
stream@(k, w)
```

where:

- **k** — the window's hop (a natural number): by how many source records the window shifts on every step,

- **w** — the window size (a non-zero integer): how many source fields a single output record contains.

A negative value of `w` means **mirrored aggregation** — the fields in the output record are laid out in reverse order relative to their arrival.

### How the output stream's interval changes

If the source stream has `W` fields per record and interval `Δ`, the output stream of the `@(k, w)` operator has:

- `|w|` fields per output record,
- an output interval `Δ_out = (Δ / W) × k`.

| Parameters | Effect |
|---|---|
| `k = \|w\|` | a tumbling window — successive windows do not overlap |
| `k < \|w\|` | a sliding window — successive windows overlap |
| `k > \|w\|` | sampling with gaps — some data is skipped |
| `k = 1, \|w\| = 1` | serialization — a multi-field record is split into single-element ones |
| `w < 0` | mirrored aggregation — the order of fields within the window is reversed |

### Typical usage patterns

```
-- serialization: 2 fields → 1 field (interval ÷ 2)
SELECT * STREAM s1 FROM A@(1,1)

-- tumbling window: windows of 4 records, no overlap
SELECT * STREAM s2 FROM A@(4,4)

-- sliding window: a 5-element window shifted by 1
SELECT * STREAM s3 FROM A@(1,5)

-- sampling: every fifth record (skip=5, window=1)
SELECT * STREAM s4 FROM A@(5,1)

-- mirrored deserialization: restoring field order
SELECT * STREAM s5 FROM s1@(2,-2)
```

### Visualizing the @ operator

Below is a schematic representation of how `source@(k, w)` behaves for a single-element stream:

```
Input data:       0  1  2  3  4  5  6  7  8  9  ...
                  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓
```

```
@(1, 3) – sliding window, hop=1, window=3:
  [0,1,2]  [1,2,3]  [2,3,4]  [3,4,5]  ...

@(3, 3) – tumbling window, hop=3, window=3:
  [0,1,2]           [3,4,5]           ...

@(5, 1) – sampling every 5 elements:
  [0]               [5]               ...

@(2,-2) – mirrored, hop=2, window=2:
  [1,0]    [3,2]    [5,4]    [7,6]    ...
```

### Examples

The subchapters below present concrete uses of the AgSe operator:

- **Serialization Example** — turning a multi-field record into a sequence of single-element records and back again via mirrored aggregation.
- **Moving Average Example** — a sliding window as the basis for a signal-averaging filter.
- **Window Types** — tumbling, sliding, and sampling on the same data stream.

We'll start by considering the serialization process using the Aggregation and Serialization operator — AgSe.

**Note**

The functionality described here is covered by the tests: `agse1`, `agse2`, `agse3`, `Pattern6`, described in the appendix Integration Tests.

### Serialization Example

Let's start by creating a file qplan3.rql with the following content:

```
DECLARE a BYTE, b BYTE STREAM A, 1 FILE 'data3.txt'
SELECT * STREAM str3 FROM A@(1,1)
```

And let's prepare a file data3.txt with the following content:

```
$ seq 0 9 | paste - -
```

```
0       1
2       3
4       5
6       7
8       9

```

The last, empty line matters and is significant. After running xretractor qplan3.rql, and in a second window xqry -s str3, we'll see something like this:

```
$ xqry -s str3
7
8
9
0
1
2
3
4
5
6
```

What we're seeing is an example of serialization. An interesting aspect of the Agse operator, in this case, is also visible in the query execution plan. We can look at it with the command:

```
$ xretractor -c qplan3.rql -f -p -d > out.dot && dot -Tsvg out.dot -o out.svg
```

In the out.svg file we'll see the following query execution plan (Fig. 52):

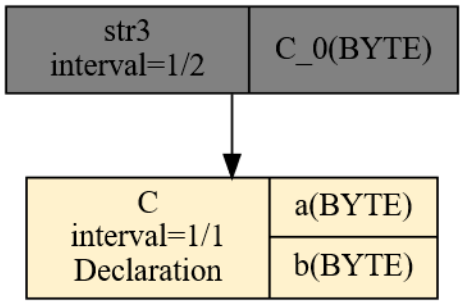



Fig. 52. Query execution plan after AGSE compilation

From a source stream, where data containing two bytes arrives every second, a data stream is created in which one byte appears every half second.

Now that we have a str3 data stream in the system, returning sequential numbers, we can use it for further transformations. Let's add the following query to the qplan3.rql file:

```
SELECT * STREAM str4 FROM str3@(2,2)
```

After running it, though, we won't see the expected original form of the data3.txt file. Instead, we'll see something like this:

```
$ xqry -s str4
2 1
4 3
6 5
8 7
0 9
```

```
2 1
4 3
6 5
```

Along the difficult path of stream processing, there are traps. This is one of them. Only once the reader looks closely will they notice that the data is mirror-flipped. Please change this query to the following form:

```
SELECT * STREAM str4 FROM str3@(2,-2)
```

Only a stream built this way will show the original form visible in the data3.txt file:

```
$ xqry -s str4
3 4
5 6
7 8
9 0
1 2
3 4
```

That minus sign in the window-width specification is the mirror flip. The window size is two, but the field sequence is built in reverse order.

Generating an image of the query plan that carries out serialization first and then deserialization, we'll see the following relationship (Fig. 53):

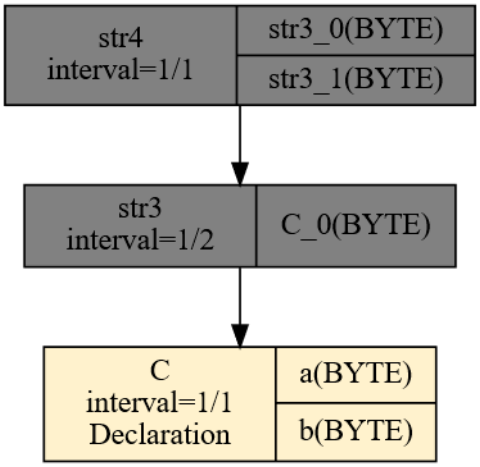



Fig. 53. SErialization and DEserialization

What's shown here is the most basic example of using the sliding-data-window operator. If we start experimenting with the hop and window size, we'll notice that we're able to create an arbitrary sliding window over the data stream, or skip some elements by building a hop larger than the window width.

**Note**

The functionality described here is covered by the tests: `agse1`, `agse2`, `agse3`, `Pattern6`, described in the appendix Integration Tests.

### Moving Average Example

The moving average is one of the simplest and most commonly used signal filters. Every output point is the arithmetic mean of the last `N` samples. The `@(1, N)` operator in RetractorDB creates exactly this kind of window: for every new measurement, the last `N` values are available.

#### Source data

Let's assume a temperature stream measured every second. The file `temp.txt` contains successive readings:

```
$ seq 10 5 60 > temp.txt
```

```
10
15
20
25
30
35
40
45
50
55
60
```

#### The RQL query

The file `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
```

### What each query does

1. `sensor@(1,5)` — creates a sliding 5-element window. Every `window5` record contains the 5 most recent temperature readings. Output interval: `1s / 1 × 1 = 1s` (hop=1, W=1 field).
2. Sum of the five fields — a classic `SELECT` over the fields `window5[0]`..`window5[4]`.
3. Dividing the sum by 5 — the result is the moving average.

### Running it

```
$ xretractor avg.rql &
$ xqry -s avg5
```

Example output (the window fills up after the first 5 samples):

```
30
35
40
45
50
```

The value `30` corresponds to the average of the first full window: `(10+15+20+25+30)/5 = 20`... note — RetractorDB does not show partial windows, so the first result to appear corresponds to the moment the window is fully saturated with data.

### Verifying the query plan

```
$ xretractor -c avg.rql -f -p -d > out.dot && dot -Tsvg out.dot -o out.svg
```

In the generated plan you can see the chain: `sensor → window5 → sumRow → avg5`. The key node is `sensor@(1,5)` — from a single-element stream arriving every second, a five-element stream is produced, continuously sliding.

### The relationship between window parameters and delay

The moving average introduces a delay of half the window length. For a window of N=5, the delay is 2 samples (2 seconds). Increasing the window:

- reduces noise (more smoothing),
- increases the delay,
- does not change the output interval (with a fixed hop k=1).

Changing the hop with a fixed window:

```
sensor@(5,5)   -- tumbling: a result every 5 seconds, no overlap
sensor@(1,5)   -- sliding:  a result every second, full overlap
```

```
sensor@(3,5)   -- partial overlap: a result every 3 seconds
```

**Note**

The functionality described here is covered by the tests: `agse1`, `agse2`, `agse3`, `Pattern6`, described in the appendix Integration Tests.

## Window Types

By choosing its two parameters, the `@(k, w)` operator lets you build every one of the classic window types used in stream processing. Below is a comparison of the patterns on one shared source stream.

### Source stream

The file `data.txt` — 12 consecutive integers:

```
$ seq 1 12 > data.txt
```

The source declaration — one record per second, one field:

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

### Tumbling window — non-overlapping windows

Hop equal to window size: `k = w`. Every input element belongs to exactly one output window.

```
SELECT * \
STREAM tumbling \
FROM src@(4,4)
```

Output interval: `1s × 4 / 1 = 4s`. Output records:

```
$ xqry -s tumbling
1  2  3  4
5  6  7  8
9 10 11 12
```

Use cases: aggregating samples over fixed time intervals (e.g. per-minute, per-hour).

### Sliding window — overlapping windows

Hop smaller than window size: `k < w`. Every input element appears in several successive windows.

```
SELECT * \
STREAM sliding \
```

```
FROM src@(1,4)
```

Output interval: `1s × 1 / 1 = 1s`. Output records:

```
$ xqry -s sliding
1  2  3  4
2  3  4  5
3  4  5  6
4  5  6  7
...
```

Use cases: moving averages, trend detection, FIR filters (as in the signal filter implementation).

### Sampling — windows with gaps

Hop larger than window size: `k > w`. Some input elements are skipped.

```
SELECT * \
STREAM sampled \
FROM src@(3,1)
```

Output interval: `1s × 3 / 1 = 3s`. Output records:

```
$ xqry -s sampled
1
4
7
10
```

Use cases: signal decimation, sample-rate reduction, diagnostics on every Nth measurement.

### Mirrored window — reversed field order

A negative `w` value reverses the order of fields in the output record, while keeping the same window size.

```
SELECT * \
STREAM mirrored \
FROM src@(2,-2)
```

Output interval: `1s × 2 / 1 = 2s`. Output records (fields in reversed order):

```
$ xqry -s mirrored
2  1
4  3
6  5
8  7
...
```

Compare this with `src@(2,2)`, which would give `1 2`, `3 4`, `5 6`... — order matching

arrival. Mirrored aggregation is necessary when reversing serialization (deserialization), as described in the serialization example.

### Summary of patterns

| Query | Window type | Interval | Record size | Overlap |
|---|---|---|---|---|
| `src@(4,4)` | tumbling | 4 s | 4 fields | none |
| `src@(1,4)` | sliding | 1 s | 4 fields | full |
| `src@(2,4)` | hop window | 2 s | 4 fields | partial |
| `src@(3,1)` | sampling | 3 s | 1 field | none |
| `src@(2,-2)` | mirrored | 2 s | 2 fields | none |

### Query execution plan

All four variants can be run at once by placing them in a single `.rql` file:

```
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
```

The query plan shows four independent branches originating from a shared `src` node. Each branch implements a different window type with no dependencies between them.

**Note**

The functionality described here is covered by the tests: `agse1`, `agse2`, `agse3`, `Pattern6`, described in the appendix Integration Tests.

## 6.5. Stream Playback

We very often think of time series as data marked with timestamps. Data stored, for example, in a file, can be processed however we like — preserving its order based on the recorded time relationships. RetractorDB has been given the ability to re-emit such a stream, preserving the recorded time relationships, as if the data were actually arriving again.

To demonstrate this, let's prepare a text file filled with data, e.g. from 30 to 45.

```
$ seq 30 45 > data.txt
```

We'll play back a file prepared this way in RetractorDB.

Next, let’s create the following file filled with queries for the system — query.rql, containing only a single declaration ending in HOLD.

```
DECLARE a INTEGER STREAM core, 1 FILE 'data.txt' HOLD
```

In one window, we run the command:

```
$ xretractor query.rql
```

In another, we issue the command:

```
$ xqry -s core
0
0
0
…
```

We’ll see a sequence of zeros …

In another window we issue the following command:

```
$ xqry -a "SELECT * STREAM ping FROM core VOLATILE”
snd: adhoc SELECT * STREAM ping FROM core VOLATILE
rcv: db OK
```

At this point, in the window showing the values from the core stream, the core values will appear:

```
$ xqry -s core
0
0
0
…
0
0
0
30
31
32
33
…
```

A recorded example below (Fig. 54):


Fig. 54. Recorded example of stream playback

# Chapter 7

# Usage Examples

This chapter presents short examples of using RetractorDB to solve concrete problems encountered when building monitoring systems.

Every example is complete — it includes a problem description, the RQL query design, how to run it, and how to interpret the results. The examples can be run on their own: the required data files and scripts are described step by step.

**Signal filtering (FIR)**

This example demonstrates how to implement a **digital FIR filter** directly within an RQL query stream, without external DSP libraries. The topic is representative of a broad class of signal-processing problems: noise filtering, frequency-band separation, time-series smoothing.

The example covers:

- designing filter coefficients in GNU Octave (the Remez algorithm, the `remez()` method),
- transferring the coefficients into a text file and loading them as a `DECLARE` stream,
- implementing discrete convolution as a set of `SELECT` queries using the sliding-window `@` operator and expansion of the _ symbol,
- real-time visualization of the filtering process using `xqry` and `gnuplot`.

The result is a working system that filters a pseudo-random signal (50 Hz) down to the 0–2 Hz band, observed live while xretractor is running.

Full description: Signal Filter Implementation

**ECG Signal Analysis (MIT-BIH)**

This example demonstrates using RetractorDB to **process clinical ECG signals** from the public MIT-BIH Arrhythmia Database (PhysioNet). It's a complex use case combining several of the system's mechanisms: multi-channel input streams, a multi-stage FIR filtering pipeline, an adaptive detection threshold, and real-time visualization.

The example covers:

- data preparation: converting MIT-BIH recordings (WFDB format) into text files compatible with RetractorDB,
- implementing the five-stage Pan-Tompkins algorithm in RQL: band-pass filter → differentiation → squaring → moving-window integration → threshold detection,
- visualizing the ECG signal and the QRS-detection result in a gnuplot window (RTL mode — newest samples on the right),
- interpreting the results: RR-interval readings, identifying arrhythmia episodes in MIT-BIH record 205.

The result is a working QRS detector processing a two-channel ECG signal (MLII + V1) at 360 Hz, implemented entirely with RQL queries, with no specialized libraries.

Full description: ECG Visualization and Arrhythmia Detection

## 7.1. Signal Filter Implementation

Problems related to digital signal processing include problems related to filtering. The goal of filtering is to separate the information contained within a signal. Usually the goal is to separate the signal from its noise.

Filters can be analog or digital. In this solution, we'll focus on digital filters. A digital filter is implemented as a sequence of operations on successive data points of the processed signal within a given time window. As a rule, when choosing a digital filter we must decide what compromises we're willing to accept. In addition, we may run into legal restrictions related to certain algorithms or methods [9].

#### Designing a filter in Octave

When designing a digital filter, we need to determine what range of frequencies we want to attenuate, and what we want to amplify or leave unaffected. We define these parameters as the stopband and the passband. One tool I know of, used for constructing digital filters, is the GNU Octave program (https://octave.org). With this tool we can generate the coefficients needed to compute a simple digital signal filter.

As an example, let's take the following values needed to construct a signal filter:

- Input signal sampling rate: 50 Hz
- Passband: 0–2 Hz
- Stopband: 5–25 Hz

An input-signal sampling rate of 50 Hz means 50 samples will appear within a second. In RetractorDB, this means the source signal should arrive at a rate of Delta = 0.02. And the defined data source should support that rate.

For these filter assumptions, the Octave program that builds the signal filter looks as follows:

```
pkg signal load
filtord = 25 % Filter length
Fs = 50;     % Sampling rate 50Hz
```

```
FNq = Fs/2;  % Nyquist frequency
F1c = 2;     % Passband 0 - 2Hz
F2c = 5;     % Stopband 5 Hz ->
F3c = 25;    % Stopband <- 25 Hz
f=[0,F1c/FNq,F2c/FNq,F3c/FNq]
m = [ 1 , 1 , 0, 0 ]
freqz ( remez(filtord,f,m) );
```

A file prepared this way should be saved to disk, or pasted directly into the Octave terminal window.

We can display the filter's floating-point parameters by issuing the command remez(filtord,f,m). We get a graphical representation of the filter by issuing the following command:

```
octave:1> [h, w] = freqz ( remez(filtord,f,m) );
subplot(2,1,1);
plot (f, m, '', w/pi, abs (h), '');
xlabel('Normalized frequency')
ylabel('gain')
grid on
subplot(2,1,2);
plot(f,20*log10(m+1e-5),'', w/pi,20*log10(abs(h)),'');
xlabel('Normalized frequency')
ylabel('gain (dB)')
grid on
```

Running the code above in Octave produces the following graphical response (Fig. 55):

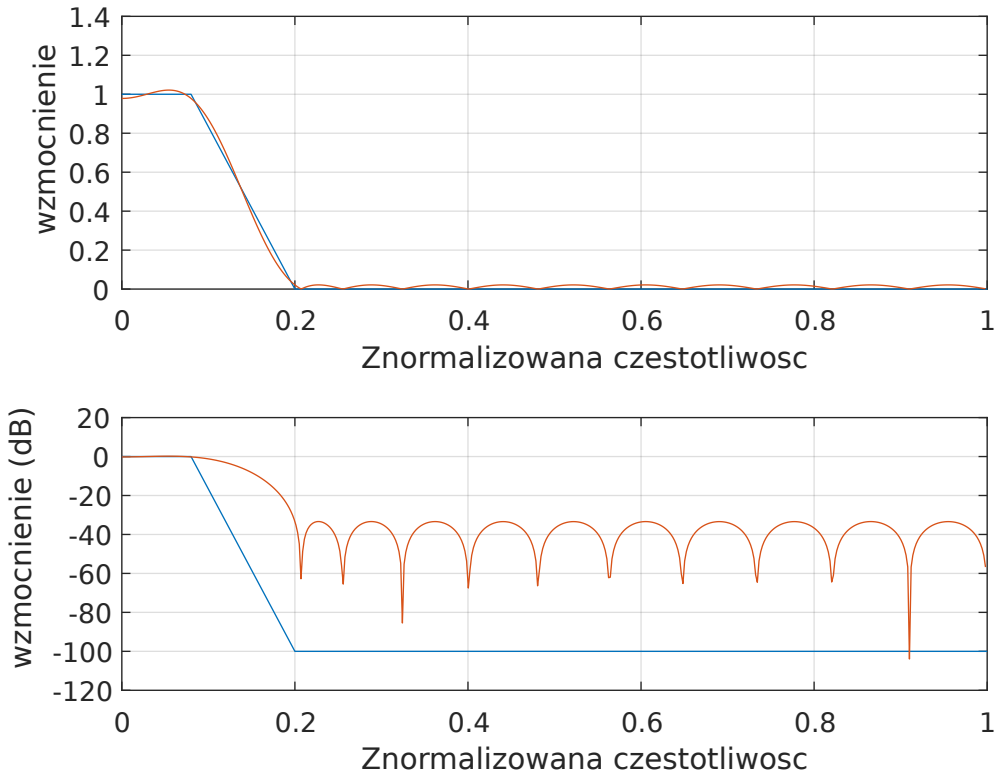



Fig. 55. Graphical representation, in the frequency domain, of the computed digital filter

On the y-axis, Octave shows the normalized frequency. The frequency range shown on the y-axis, from 0 to 1, corresponds to a frequency range of 0Hz to 25Hz. On the x-axis, the first plot shows the linear gain, the second the same quantity but on a logarithmic scale.

The filter's parameters can be displayed with the command:

```
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
…
```

To get fixed-point parameters for a 16-bit filter, run the command:

```
octave:12> floor(remez(filtord,f,m) * 32767)
ans =
```

```
   -140
   -661
   -488
   -596
   -460
```

### Implementation in RetractorDB

We should transfer the values obtained into a text file named filterremez.txt.

For testing purposes, we'll take the source signal from a pseudo-random number generator. We'll pull the ephemeral data directly from the source at a rate of 50Hz.

The initial part of the query.rql file, containing the source declarations for RetractorDB, looks as follows:

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

DECLARE data BYTE \
STREAM source, 0.02 \
FILE '/dev/urandom'
```

In the next part we'll find the commands that build the signal-processing pipeline.

```
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
```

Here we see 4 queries. Reviewing the chapter on underscore symbol expansion, it shouldn't surprise us that trying to preview the compilation result of this file will scroll through several screens. We can get a quick-to-analyze preview of the process taking place by issuing the command:

```
$ xretractor -c query.rql -p -d > out.dot && dot -Tsvg out.dot -o out.svg
```

We'll see the following picture (Fig. 56):

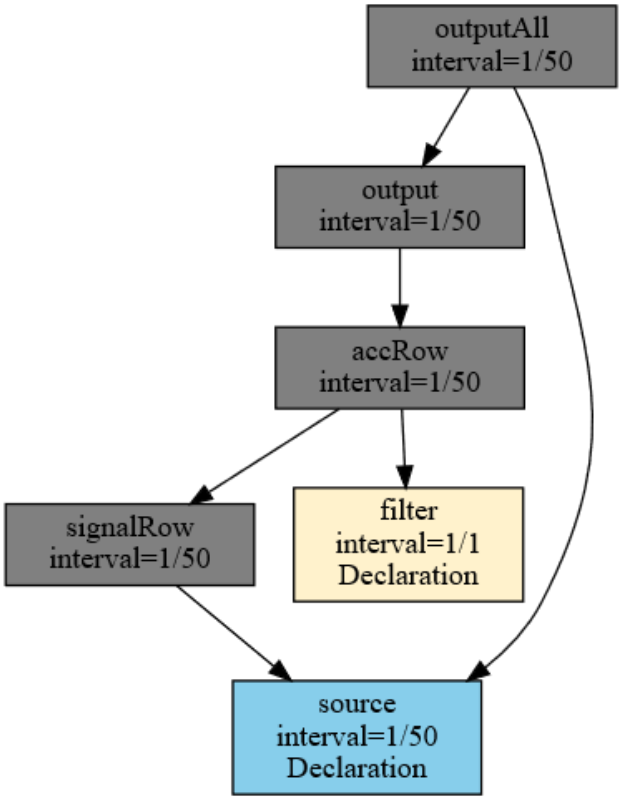



Fig. 56. Dependency between the processed data streams while carrying out the signal filter

**Running it**

Trying to inspect the contained fields and data types would expand the generated figure so much that it's impossible to include the generated result here without losing readability.

To watch the signal-processing pipeline in real time, we should issue the following sequence of commands:

- in the first window, start the server process that processes the collected data. The files query.rql and filterremez.txt should be present in this directory, with the command:

```
$ xretractor query.rql
```

- in the second window, issue the following command:

```
$ xqry -s outputAll -p 50:256 | gnuplot
```

On screen we should see the following chart, running from left to right, continuously filled with data (Fig. 57):

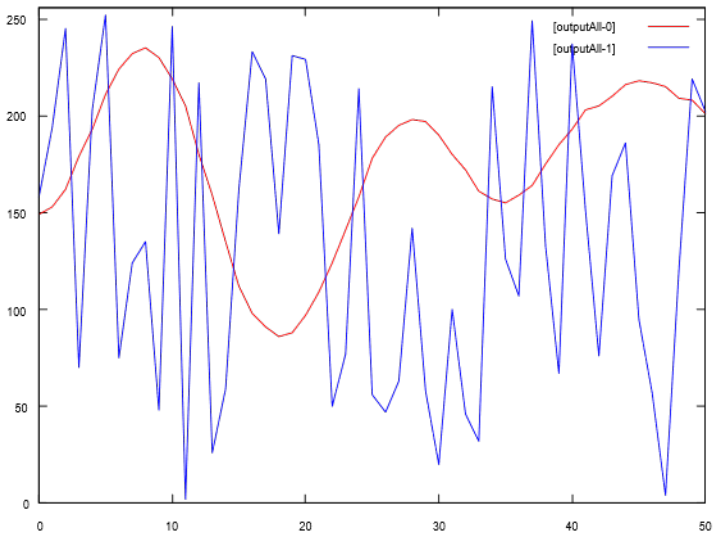



Fig. 57. Signal filtering carried out inside RetractorDB

In Fig. 57 we see two charts overlaid on each other. The more erratic one — shown on screen as a blue line with a lot of variability — is the visualization of the input signal. Data taken from the pseudo-random number generator at a rate of 50 samples per second. And the second chart, wrapping around the input data — shown on screen in red, smoother, flowing around it — is exactly the data filtered by the signal filter we built. A signal whose passband has been restricted to 0–2Hz (low frequencies) and stopped in the region of 5–25Hz (high frequencies). Figuratively speaking, we've isolated the bass line.

Keep in mind that, on screen, this chart scrolls to the right very quickly, showcasing the current data-processing capabilities carried out in RetractorDB.

A screen recording during the processing run is shown in Fig. 58:

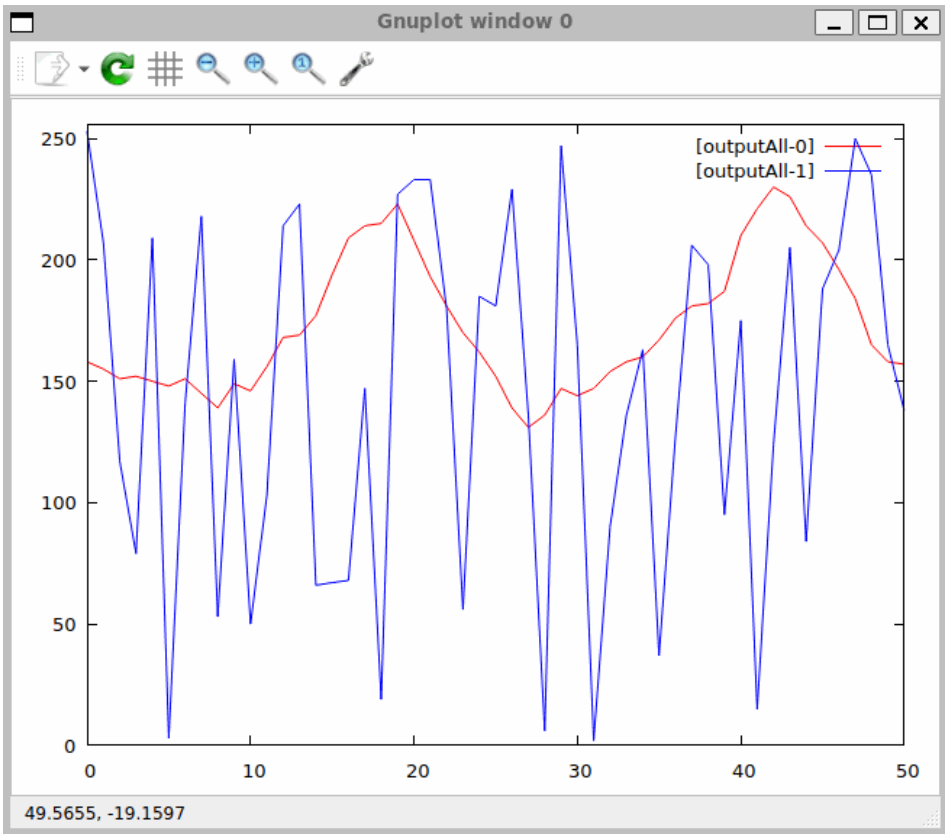



Fig. 58. Animation of the real-time signal-filtering process

**Note**

The functionality described here is covered by the test: `dsp`, described in the appendix Integration Tests.

## 7.2. ECG Visualization and Arrhythmia Detection — the MIT-BIH Database

### Data source — the PhysioNet MIT-BIH Arrhythmia Database

The MIT-BIH Arrhythmia Database is a publicly available collection of electrocardiogram recordings published by PhysioNet at:

```
https://physionet.org/content/mitdb/1.0.0/
```

It contains 48 half-hour, two-channel recordings collected from 47 patients at Beth Israel Hospital in Boston between 1975 and 1979. The recordings were manually annotated by at least two independent cardiologists and are widely used in research on automatic arrhythmia detection.

### Record 205

The example uses record **205** — a recording from a 59-year-old man treated with Digoxin and Quinaglute. The record contains episodes of ventricular tachycardia (VT) and is often cited in the literature as diagnostically challenging, due to two morphologically distinct forms of premature ventricular contractions (PVC).

Recording parameters:

| Parameter | Value |
|---|---|
| Duration | ≈ 30 min |
| Sampling rate | 360 Hz |
| Number of samples | 650,000 |
| Channel 1 (MLII) | Modified limb lead II |
| Channel 2 (V1) | Precordial lead V1 |
| Resolution | 12 bits, gain 200 LSB/mV, zero point 1024 |

Raw values are stored as unitless integers (so-called ADC values). Conversion to millivolts:

$$\text{mV} = \frac{\text{ADC} - 1024}{200}$$

The range of actual values in the rec205 file falls between 589–1315 (MLII) and 718–1106 (V1), corresponding to a signal amplitude of roughly ±1.5 mV.

## Data preparation

The original recording files (`205.hea`, `205.dat`, `205.atr`) are provided in the MIT-BIH format and need to be converted into a binary format recognized by RetractorDB.

### The MIT-BIH format 212

The signal in the `205.dat` file is packed 12-bit in format 212: every three bytes store two consecutive samples of both channels according to the scheme:

```
[B0][B1][B2] → MLII = B0  | ((B1 & 0x0F) << 8)
               V1   = B2  | ((B1 >>  4)  << 8)
```

The values are 12-bit signed (range –2048..2047).

### Converting to the RetractorDB format

The script `examples/ecg/mitbih2rdb.py` reads the `205.hea` header, decodes the sample pairs, and writes them as little-endian `int32` records into the `rec205` file:

```
650,000 records × 2 fields × 4 bytes = 5,200,000 bytes
```

At the same time, the script generates an RQL script that plays back the signal (`rec205-replay.rql`). The descriptor file `rec205.desc` is created by `build.sh`.

The whole preparation process is run with a single command from the project's root directory:

```
bash examples/ecg/build.sh
```

The result is three files in the `examples/ecg/rec205/` directory:

| File | Generated by | Description |
|---|---|---|
| `rec205` | `mitbih2rdb.py` | Binary data (int32 LE) |
| `rec205.desc` | `build.sh` | Stream descriptor |
| `rec205-replay.rql` | `mitbih2rdb.py` | RQL playback script |

### The RQL query

The file `rec205-replay.rql` defines two streams:

```
DECLARE MLII INTEGER, V1 INTEGER STREAM ecg, 1/360 FILE 'rec205'

SELECT ecg.MLII, ecg.V1 STREAM s205out FROM ecg VOLATILE
```

The `STREAM ecg, 1/360` clause sets the time interval of a single sample to 1/360 s, matching the actual sampling rate of 360 Hz. The `TYPE DEVICE` clause in the descriptor causes the `rec205` file to be read sequentially in a loop (after the last sample, reading returns to the beginning), enabling continuous playback of the recording.

The output stream `s205out` is declared `VOLATILE`, so it is not written to disk — the data only reaches the consumer process (`xqry`).

### On-screen visualization

The `ecg` target in the build system is used to display the chart in real time. It runs the script `scripts/xplot.sh`, which starts `xretractor` in the background, and then pipes the data stream through `xqry` into `gnuplot`.

```
## from build/Debug
ninja ecg
```

The invocation CMake expands this into:

```
scripts/xplot.sh s205out rec205-replay.rql 720,560,1360 --gnuplot-rtl
```

Meaning of the parameters:

| Parameter | Meaning |
|---|---|
| `s205out` | The name of the output stream |
| `rec205-replay.rql` | The query file |
| `720` | The data window width (samples visible at once) |

| Parameter | Meaning |
|---|---|
| `560,1360` | The Y-axis range (ADC values matching the actual signal) |
| `--gnuplot-rtl` | Newest samples on the right, chart scrolls right to left |

The `--gnuplot-rtl` option is an `xqry` parameter that reverses gnuplot's X axis (`set xrange [720:0]`). The effect is that the freshest samples appear on the right side of the window, and older ones scroll to the left — similar to the classic ECG printout on paper tape.

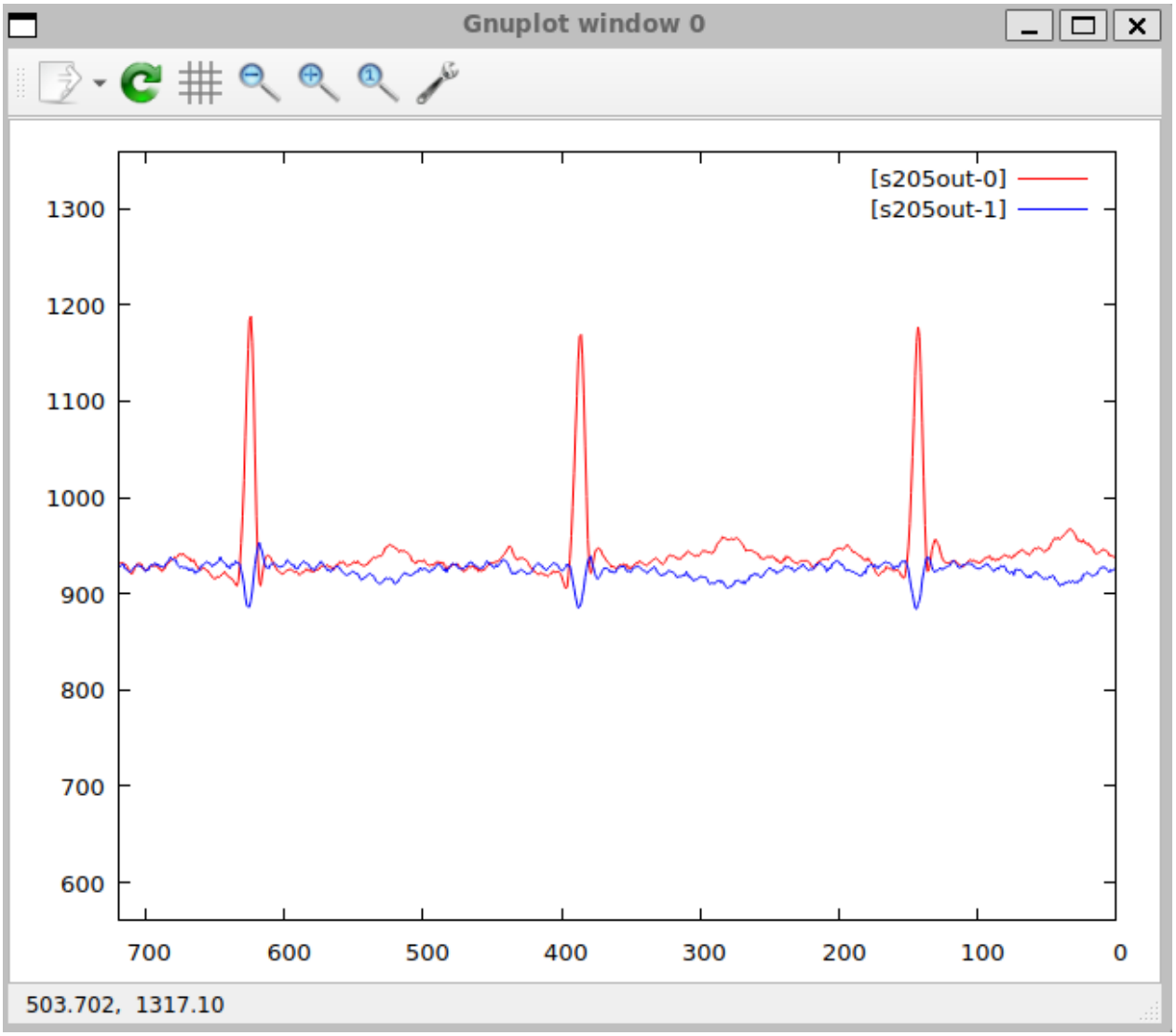



Fig. 59. View of the gnuplot window with the ECG signal being played back (record 205)

The window shown in Fig. 59 displays 720 samples, i.e. exactly 2 seconds of signal at 360 Hz, matching the typical width of a single ECG strip used in diagnostics.

## QRS Detection and Arrhythmia Identification

### Context — the Pan-Tompkins algorithm

QRS-complex detection is the foundation of automatic ECG analysis. The QRS complex represents ventricular depolarization and corresponds to every heartbeat, visible as a sharp spike in the signal. Knowing the positions of the QRS complexes in time, RR intervals can be computed, and from them, basic rhythm disturbances can be identified:

| Measure derived from QRS | Use |
|---|---|
| RR intervals | Heart rate (HR), VT, bradycardia |
| RR variability (HRV) | Autonomic nervous system, event prediction |
| QRS morphology | Distinguishing PVC from a normal rhythm, APC |
| QRS duration | Bundle branch block (BBB) |

The Pan-Tompkins algorithm (1985) is a classic, five-stage, pipelined digital-signal-processing algorithm implemented with FIR filters. RetractorDB implements it directly as an RQL query stream, without specialized DSP libraries.

### Generating the signal filters (coef)

The algorithm requires two sets of FIR coefficients, stored as text files (`bp_coef.txt`, `d_coef.txt`). They are generated once, with Python scripts, before running detection.

**The band-pass filter — `gen_bp_coef.py`** Step 1 of the algorithm requires a filter that cuts out noise and artifacts outside the QRS band. A passband of 5–15 Hz at fs = 360 Hz produces a response containing the QRS morphology, while attenuating baseline wander (< 5 Hz) and muscle noise (> 15 Hz).

The design method is a **windowed sinc**:

```
h_bp[n] = (h_lp2[n] − h_lp1[n]) · w[n]
```

where:

- `h_lp[n] = 2·fc·sinc(2·fc·(n−M))` — the ideal low-pass filter
- `w[n] = 0.54 − 0.46·cos(2πn/(N−1))` — the Hamming window, damping Gibbs effects
- `M = (N−1)/2 = 12` — the filter's center point (group delay = 12 samples)

Parameters:

| Parameter | Value |
|---|---|
| Filter length N | 25 coefficients |
| Lower cutoff $fc_1$ | 5 Hz (normalized 5/360) |
| Upper cutoff $fc_2$ | 15 Hz (normalized 15/360) |

| Parameter | Value |
|---|---|
| Integer scale | ×1000 (divided /1000 in RQL) |

Running the script:

```
cd examples/ecg/rec205
python3 gen_bp_coef.py
## Saved 25 coefficients to bp_coef.txt
## Coefficients: [-2, -2, -1, 0, 3, 8, 14, 23, 32, 41, 49, 54, 56, ...]
## Sum (DC gain): 5 / 1000 = 0.0050
```

The coefficients are symmetric about the center (n=12), confirming the filter's linear phase — an essential property when analyzing ECG, since it guarantees no phase distortion of the QRS morphology.

**The differentiating filter — `gen_d_coef.py`** Step 2 of the algorithm applies a filter that emphasizes the steep edges of the QRS. Pan and Tompkins proposed a 5-point derivative estimator:

```
y[n] = (1/8T) · (−x[n−4] − 2·x[n−3] + 2·x[n−1] + x[n])
```

Coefficients (from oldest to newest sample):

```
h = [−1, −2, 0, 2, 1]
```

Filter properties:

| Property | Value |
|---|---|
| Sum of coefficients | 0 (zero DC gain — eliminates offsets) |
| Maximum response | f ≈ 10–25 Hz (the QRS-edge range) |
| Scale factor (1/8T) | 360/8 = 45 Hz (ignored — does not affect detection) |

```
cd examples/ecg/rec205
python3 gen_d_coef.py
## Saved 5 coefficients to d_coef.txt
## Coefficients: [-1, -2, 0, 2, 1]
## Sum (DC gain): 0  (should be 0)
```

### Implementing the pipeline in RQL — `rec205-detect.rql`

The file `rec205-detect.rql` implements the complete five-stage pipeline for both ECG channels (MLII and 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'
```

```
## Extracting the channels
SELECT ecg.MLII STREAM mlii FROM ecg VOLATILE
SELECT ecg.V1   STREAM v1   FROM ecg VOLATILE

## 1. Band-pass filter (5-15 Hz) — 25-tap FIR convolution
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. Differentiation — 5-tap FIR convolution
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. Squaring (/1000 prevents int32 overflow)
SELECT d_out[0]*d_out[0]/1000   STREAM sq_out   FROM d_out        VOLATILE

## 4. Moving-window integration over 30 samples (~83 ms)
SELECT *                        STREAM mwi_win  FROM sq_out@(1,30) VOLATILE
SELECT mwi_win[0]               STREAM mwi      FROM mwi_win.avg   VOLATILE

## 5. Adaptive threshold — 2x moving average over 180 samples (0.5 s)
SELECT *                        STREAM mwi_long FROM mwi@(1,180)  VOLATILE
SELECT mwi_long[0]              STREAM mwi_thr  FROM mwi_long.avg VOLATILE

## Output: MLII centered, V1 centered, detection signal ×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
```

**Rationale for the parameters** The `@(1,25)` operator creates a sliding window of 25 samples, while `[_]` and `sumc` implement discrete convolution — see the chapter Underscore Symbol Processing.

The `/1000` division in step 3 compensates for the integer scale of the coefficients — without it, the product `d_out × d_out` would exceed the `int32` range (2,147,483,647) for typical ECG amplitudes.

The output expression `(mwi[0]-mwi_thr[0]*2)*5` implements the adaptive threshold: the value is positive only when the MWI envelope exceeds twice the current moving average — indicating a detected QRS. The `×5` multiplier scales the detection signal to a range visually comparable to the raw ECG on the chart.

### Running it — ninja ecg-detect-qrs

The process is started with a single command from the `build/Debug` directory:

```
cd build/Debug
ninja ecg-detect-qrs
```

CMake expands this target into the command:

```
scripts/xplot.sh detect_out rec205-detect.rql 720,-400,400 --gnuplot-rtl
```

Meaning of the parameters:

| Parameter | Meaning |
|---|---|
| `detect_out` | The name of the output stream (3 fields) |
| `rec205-detect.rql` | The query file with the pipeline above |
| `720` | Window width: 720 samples = 2 seconds at 360 Hz |
| `-400,400` | Y-axis range in ADC units (≈ ±2 mV) |
| `--gnuplot-rtl` | Newest samples on the right (right-to-left) |

The `xplot.sh` script starts `xretractor` in the background (compiling and executing the queries), then pipes the `detect_out` stream through `xqry` into `gnuplot` in continuous mode. The `gnuplot` window refreshes with every new batch of samples.

**Figure description — the gnuplot window**

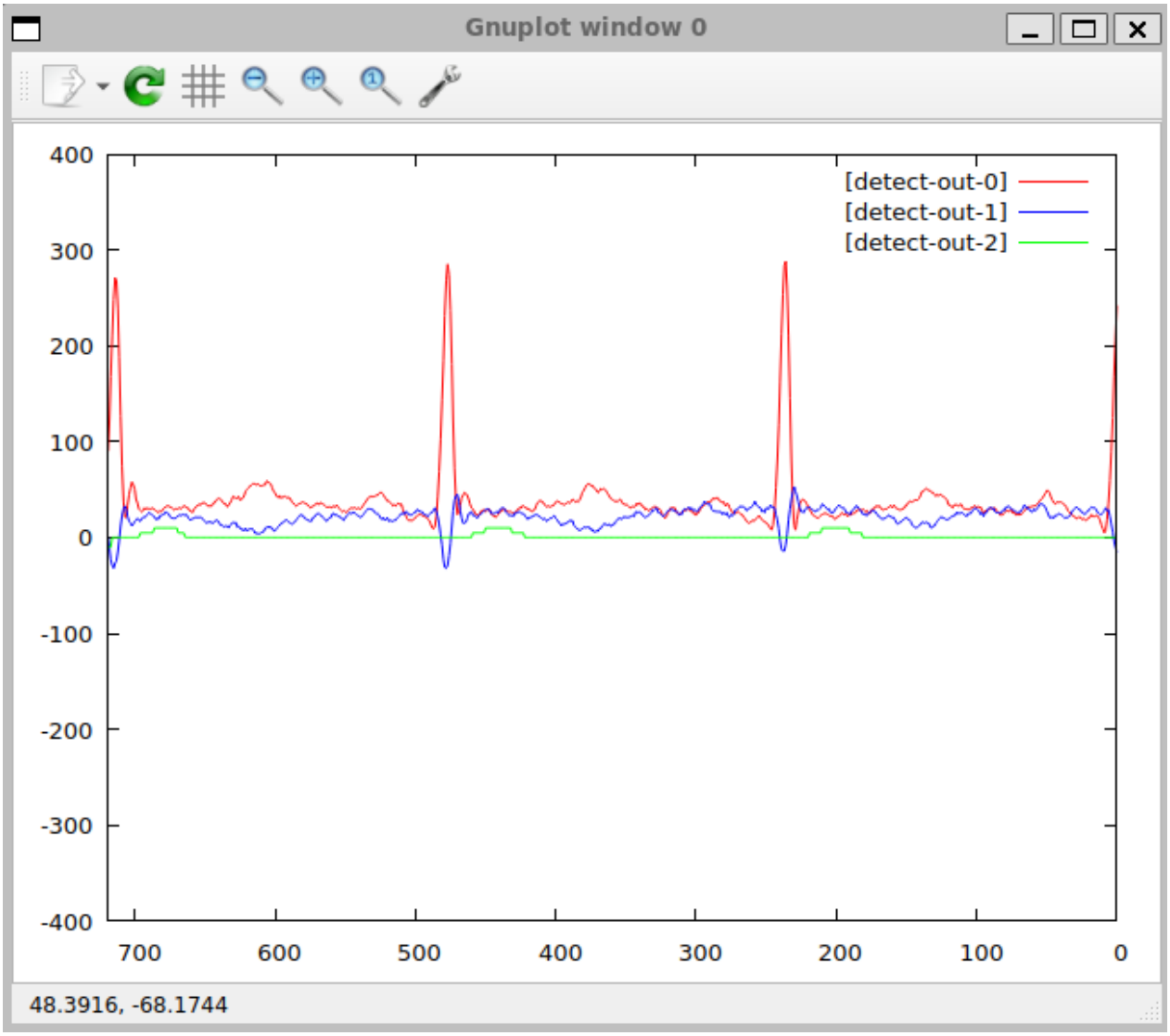



Fig. 60. The gnuplot window from running ninja ecg-detect-qrs — MIT-BIH record 205, 720 samples (2 s), RTL

In Fig. 60, three signals are visible, corresponding to the three fields of the `detect_out` stream:

**[detect-out-0] red line — MLII centered (mlii − 900)**

The raw ECG signal from lead MLII, shifted by the 900 ADC baseline point so that the zero axis corresponds to the isoline. Two sharp spikes (amplitude ≈ 280 ADC ≈ 1.4 mV) around samples 520 and 350 from the right edge represent two consecutive QRS complexes. The clear QRS morphology, with a dominant R peak, confirms the band-pass filter is working correctly — noise has been suppressed while the peak retained its amplitude.

**[detect-out-1] blue line — V1 centered (v1 − 900)**

The signal from lead V1 of the same recording. QRS morphology in V1 is, as a rule, less pronounced than in MLII, which is visible in the figure — the blue signal shows a smaller R-peak amplitude at similar QRS time positions. Having both channels available at once allows distinguishing supraventricular contractions (APC) from ventricular ones (PVC), since ventricular QRS complexes show a distinct morphology in V1.

**[detect-out-2] green line — the QRS detection signal ((mwi − 2·mwi_thr) × 5)**

The algorithm's output signal. A **positive** value means a detected QRS complex — the moving-window-integration envelope exceeded twice the adaptive threshold. In the figure, two clear positive pulses are visible, coinciding in time with the QRS peaks on the MLII channel. Between beats, the line stays close to zero or slightly below — confirming the detector's specificity.

The gap between the two visible QRS complexes is approximately 170 samples, which at 360 Hz gives:

```
RR ≈ 170 / 360 ≈ 0.47 s  →  HR ≈ 127 bpm
```

This value falls within the range of ventricular tachycardia (VT, 79–216 bpm) recorded in record 205, suggesting that the visualized fragment of the recording comes from one of the 6 VT episodes noted by the MIT-BIH cardiologists.

### Process flow diagram

The diagram below (Fig. 61) shows the complete data flow from the raw MIT-BIH recording to arrhythmia identification, indicating where RetractorDB carries out the Pan-Tompkins algorithm, and its relationship to classic arrhythmia-recognition methods:

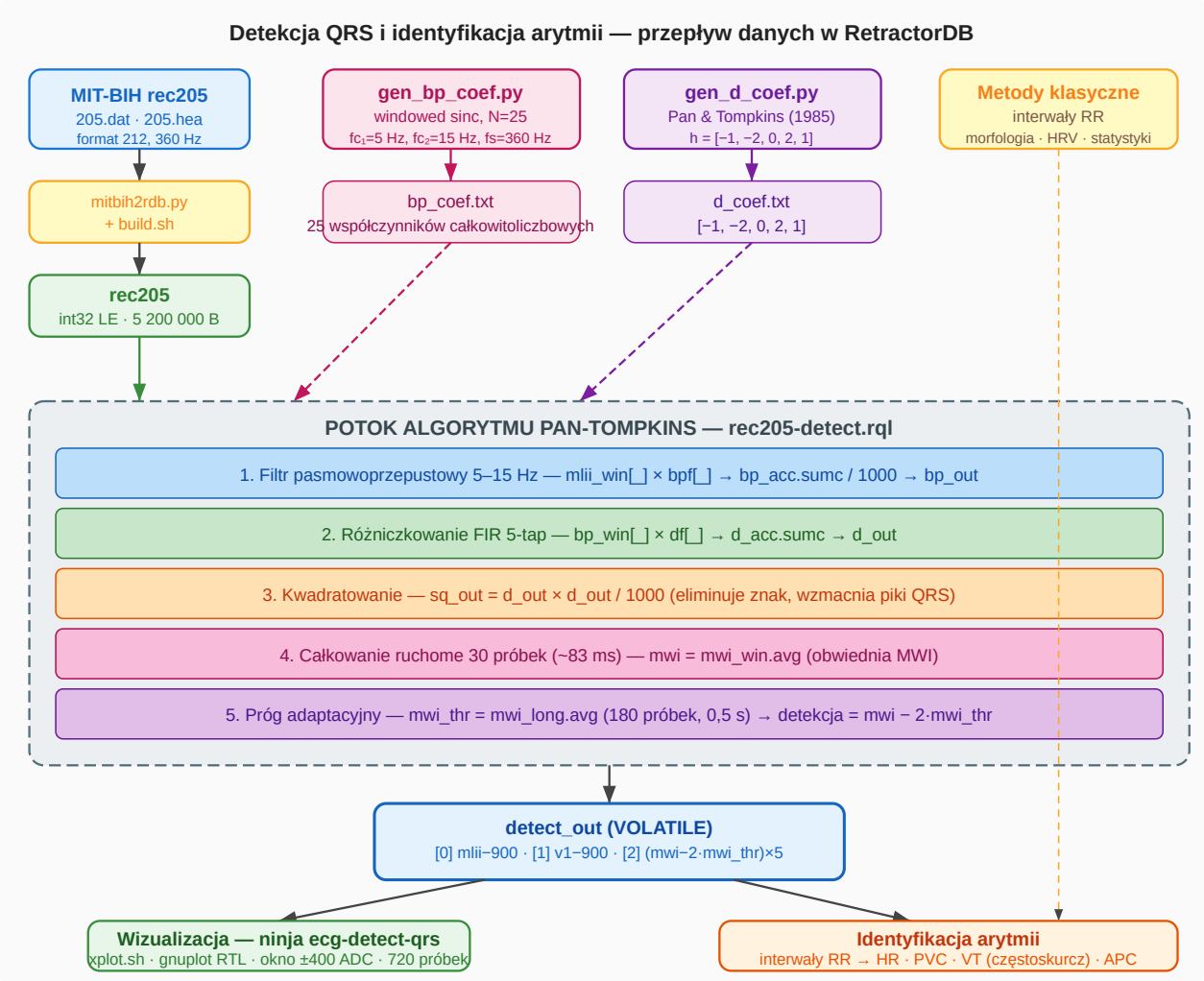



Fig. 61. Data flow — from the MIT-BIH recording, through the Pan-Tompkins pipeline in RQL, to visualization and arrhythmia identification

The right branch of the diagram — **Arrhythmia identification** — represents classic post-QRS-detection analysis methods, which can be built as further RQL queries layered on top of the `detect_out` stream:

| Method | Description | Relationship to QRS |
|---|---|---|
| RR intervals | Time between consecutive QRS complexes → HR | directly from detection positions |
| HRV (variability) | Standard deviation of RR | RR-stream statistics |
| PVC classification | QRS width > 120 ms, V1 morphology | width of the `mwi` window |
| VT detection | Sequence of ≥ 3 PVCs with HR > 100 bpm | RULE on the HR+PVC stream |

| | | |
|---|---|---|
| APC detection | An early, narrow QRS preceding a pause | MLII vs V1 morphology |

RetractorDB provides `RULE` operators and windowed aggregates (`.avg`, `.sumc`), which make it possible to implement the methods above in the same RQL query language, without leaving the system’s environment. QRS detection is the first and necessary step in this hierarchy.

# Chapter 8

# Appendices

The appendices contain documents not directly related to the system's construction, but which describe the motivation behind design decisions, tool documentation, and supporting material for people deploying or extending the system.

**System Origin**

A description of the historical circumstances that led to RetractorDB's creation. The starting point is the author's experience building a neonatal monitoring system in the early 2000s — running into the limitations of relational databases when recording high-granularity signals, attempts based on the stream-processing systems of the time, and the evolution toward a dedicated time-series processing engine. The chapter also explains where the name "Retractor" comes from — a reference to a group of surgical instruments that separate and join tissue structures, treated here as an analogy for operations on data streams.

Full description: System Origin

**Further Development Directions**

An outline of potential extensions to the algebra underlying RQL. The main thread is the search for a generalization to complex numbers — a direct application of Gaussian integers, assuming the computational basis would be rational numbers, did not produce the expected results, due to the nature of the modulus (the modulus of a complex number with rational components is real, not rational). An alternative is **Eisenstein integers** — a threefold-symmetric counterpart to Gaussian numbers, whose modulus preserves rational properties. The chapter includes a derivation of their definition and a preliminary analysis of their applicability to time-series algebra.

Full description: Further Development Directions

**RQL Syntax Highlighting**

RetractorDB query files (extension `.rql`) have dedicated syntax-highlighting definitions for three environments:

- **Visual Studio Code** — the `rql-vscode` extension, installed from the GitHub repository,

- **Vim** — the files `syntax/rql.vim` and `ftdetect/rql.vim`, installed via `scripts/buildrdb.sh vimsyntax` or manually into `~/.vim/`,
- **bat / batcat** — a Sublime Text 3 format definition, installed via `scripts/buildrdb.sh batsyntax`.

Each environment recognizes RQL keywords (`SELECT`, `DECLARE`, `RULE`, `STREAM`, …), data types, comments, string literals, and numeric values.

Full description: Syntax Highlighting

**Command-Line Options**

Complete command-line flag documentation for all three of the system's tools:

| Tool | Role |
|---|---|
| `xretractor` | The main processing process: compiles RQL queries and executes the plan |
| `xqry` | Client: queries a running `xretractor` via shared memory |
| `xtrdb` | Inspection tool: analyzes binary artifacts and metadata |

Each tool is described in its own subchapter, with example invocations and explanations of the individual switches.

Full description: Command-Line Options

**Integration Tests**

A catalog of all the system's integration tests, with a description of the functionality each one verifies. Integration tests run the actual binaries (`xretractor`, `xqry`, `xtrdb`) and compare their results against patterns — unlike GTest unit tests, which test isolated library classes.

The tests are split into two sets:

- **`IntegrationTest_serial`** — require a running IPC server; run sequentially (one after another) due to a shared lock file and Boost memory segments,
- **`IntegrationTest_parallel`** — query compilation and file inspection without an IPC server; can run in parallel.

Running them: `ninja test` or `ctest -R <name> -V` in the `build/Debug/` directory.

Full description: Integration Tests

## 8.1. Command-Line Options

RetractorDB consists of three command-line tools, each playing a distinct role in the system's architecture:

| Tool | Role |
|---|---|
| `xretractor` | The main processing process: compiles RQL queries and executes the plan |
| `xqry` | Client: queries a running `xretractor` via shared memory |
| `xtrdb` | Inspection tool: analyzes binary artifacts and metadata |

Each tool is described in its own subchapter.

### xretractor

The `xretractor` program is RetractorDB's core process. It compiles files containing RQL queries and executes the data-processing plan. It's built to run autonomously as a systemd daemon process.

#### Modes of operation

`xretractor` starts in one of two modes:

| Mode | Description |
|---|---|
| **Processing** | Default — compiles queries and starts the query-execution loop |
| **Compile only** `-c` | Compiles queries without starting the loop; allows visualizing the plan |

Calling `-h` shows a different option list depending on the mode — option shorthands overlap, so pay attention to which mode a given option applies in.

---

#### Processing mode (default)

```
$ 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 wakeup)
  -m [ --tlimitqry ] arg (=0) query limit, 0 - no limit
```

**Processing-mode options**

| Option | Meaning |
|---|---|
| `help` | Displays the help text. The list differs depending on the mode (with or without `-c`). |
| `onlycompile` | Switches the tool into “compile only” mode. The query-execution loop is not started. |
| `queryfile` | The name of the query file to compile and run. |
| `quiet` | Skips displaying results on screen. Processing runs normally, but the result presenter isn’t started. |
| `status` | Checks whether another `xretractor` process is running, or has left behind lock files preventing multiple instances. |
| `verbose` | An increased-verbosity mode — shows stream parameters. A leftover from the development phase; likely to be kept. |
| `xqrywait` | Compiles the queries and holds off the processing loop until the first query arrives from an `xqry` process. Required when using `-m N` at the same time in scripts and tests: without this flag, the server may process all N cycles before the client manages to connect, resulting in no data and `xqry` waiting until it times out. The first command received from `xqry` (e.g. `-d` or `-s`) unblocks the processing loop. |
| `noanykey` | No keypress interrupts the processing loop. Without this option, pressing any key stops the system. |
| `realtime` | Enables real-time scheduling: `SCHED_FIFO`, `mlockall`, and absolute sleep for the processing thread. Requires `CAP_SYS_NICE` and `CAP_IPC_LOCK` capabilities (or root). Recommended in production environments requiring deterministic response time. |
| `tlimitqry` | Limits the number of iterations in the query-execution loop. A value of `0` means no limit. |

---

**Compile-only mode (`-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
```

In this mode, options for creating diagrams and diagnostic dumps, described in more detail elsewhere in this work, are available.

**Visualization and diagnostic options**

| Option | Meaning |
|---|---|
| `help` | Displays the help text (identical to processing mode; the list differs depending on the mode). |
| `onlycompile` | On — this table describes the options that apply while the `-c` flag is active. |
| `queryfile` | The name of the query file to compile. |
| `quiet` | Tests only the compilation process itself, without presenting results. The other presentation options are not started. Included for development purposes. |
| `dot` | Creates a text file in DOT format describing the hierarchical structures produced by the compiler. The file can be passed to the Graphviz tool to generate a graphical description of the dependencies. |
| `csv` | Exports the hierarchical data structures to a CSV file (comma-separated values). |
| `fields` | Adds, to the DOT graph, the fields and their types for each data stream. |
| `tags` | Adds, to the DOT graph, the internal-language programs that build the fields of each query. Must be called together with `fields` — it visually links the fields to their programs. |
| `streamprogs` | Adds, to the DOT graph, the stream-algebra programs that build each query’s streams. |
| `rules` | Adds alerting rules to the graph. |
| `hideruleprog` | Hides the programs describing the alerting conditions (used together with `rules`). |
| `transparent` | Generates the graph with a transparent background. |

| Option | Meaning |
|---|---|
| `diagram` | Generates marble diagrams. The argument takes the form `type:cycle_count`: `type` (`0` or `1`) determines whether the diagrams show timestamps; `cycle_count` sets the number of cycles shown in the diagram. |

---

### Version Information

At the end of every help message, a line with build information is displayed:

```
Branch: issue_31-doc:2707ce0,
Code compiler: GNU Ver. 13.3.0,
Build time: 2512211449,
Type: Debug
```

| Field | Meaning |
|---|---|
| `Branch` | The repository branch name and the commit hash the program was built from |
| `Code compiler` | The GCC compiler version used for the build |
| `Build time` | The compilation date and time, in `YYMMDDHHMM` format (here: December 21, 2025, 14:49) |
| `Type` | The build type: `Debug` or `Release` |

The next line indicates the log file location:

```
Log: /tmp/xretractor.log
```

The file `/tmp/xretractor.log` records the history of invocations and the system's internal events. In a production environment, this file should be cleaned up or rotated regularly.

The last line contains MIT license information, which allows safe use of the code in corporate applications.

### xqry

The `xqry` program is an integral part of RetractorDB. It shares a memory area (Boost IPC) with `xretractor`, used for communication. It's used to query the running processing process, receive results from the query loop, and control the server's operation.

Unlike `xretractor`, `xqry` can be run in multiple instances at once.

### Running it

```
$ 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
```

---

**Receiving data from streams**

| Option | Meaning |
|---|---|
| `-s` / `select arg` | Receives data from the given stream exposed by `xretractor`. |
| `-t` / `detail arg` | Shows detailed information about a stream: its name, delta, query text, and field list with types (YAML). |
| `-a` / `adhoc arg` | Attaches a query to the system while it's running (ad hoc mode). |
| `-m` / `tlimitqry arg` | Limits the number of results received. A value of `0` means no limit. Especially useful with the `-k` option. |
| `-n` / `null` | Skips rows where every field is null. Useful for streams with measurement gaps — it removes noise from the output without client-side filtering. |

Example response for the `detail` option:

```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
```

---

### Diagnostics and server control

| Option | Meaning |
|---|---|
| `-l` / `hello` | Verifies the communication channel with `xretractor` (a diagnostic ping). |
| `-k` / `kill` | Requests that the `xretractor` process stop. |
| `-d` / `dir` | Lists all queries running in `xretractor`, in text format. |
| `-y` / `diryaml` | Lists all queries in YAML format. |
| `-w` / `wait-server` | Polls every 100 ms whether `xretractor` is available (up to 30 s), and once confirmed, carries out the requested command. Allows `xqry` to be started reliably in startup scripts and containers when process start order isn't guaranteed. Only checks IPC availability — it doesn't send any command to the server, and doesn't trigger data processing. |

---

### Output formats

`xqry` supports four data-presentation formats. The format is chosen with a flag — it can be combined with the `select` option.

| Option | Format | Use case |
|---|---|---|
| `-r` / `raw` | Text | The default. Undecorated data — useful for scripts and piping. |
| `-g` / `graphite` | Graphite | The `metric value timestamp` format — ready to send to Graphite. |
| `-f` / `influxdb` | InfluxDB | InfluxDB line protocol — ready to import into a time-series database. |
| `-p` / `gnuplot` `x,y` or `x,ymin,ymax` | Gnuplot | Aggregates for direct feeding into `gnuplot`. The argument `x,y` gives the time axis and the value; `x,ymin,ymax` additionally restricts the Y-axis range. The separator can be `,` or `:`. |

---

### Controlling the receive mode

| Option | Meaning |
|---|---|
| `-h` / `help` | Displays the help text. |
| `-c` / `needctrlc` | In normal mode, any keypress stops receiving data. This option requires using `Ctrl+C` instead. |

---

---

### Launch pattern in scripts

When using `xretractor -m N` (a limited number of cycles), there's a risk of a race: the server may process all the data before the client manages to connect. Guaranteed pattern:

```
### Server side: -x causes processing to be held off until
### the first command arrives from xqry
xretractor query.rql -m 100 -k -x &

### Client side: -w checks IPC readiness without sending commands,
### so it doesn't accidentally trigger processing
xqry -w -s stream -m 10
```

The `-w` and `-x` flags are complementary:

| Flag | Tool | Role |
|---|---|---|
| `-w` / `wait-server` | `xqry` | Waits for the server's IPC to become ready before sending a command |
| `-x` / `xqrywait` | `xretractor` | Holds off processing until the first command arrives from a client |

Without `xretractor -x`, with fast file-based streams, the data may be processed in full before the client connects — `xqry` will wait for data that never arrives.

---

### Version information

The information at the bottom of the help listing is identical to `xretractor`'s — it includes the repository branch name, compiler version, build time, and the log file path (`/tmp/xqry.log`). A description of the format can be found in the chapter xretractor — Version Information.

## xtrdb

The `xtrdb` program is an interactive tool for analyzing artifacts and substrates written by RetractorDB. It mostly works in interactive mode (a REPL), but it also offers a few startup options (e.g. `--help`, `--noprompt`, `--storagemap`).

**Warning**

Running `xtrdb` blocks a concurrently running `xretractor` — stop the server, or wait for the system to finish, before using `xtrdb`. The tool detects the lock itself and reports an error if `xretractor` is running.

---

### Running it

```
$ xtrdb                    # interactive mode (with a prompt)
$ xtrdb -n                 # batch mode (no prompt, no "ok")
$ xtrdb --noprompt         # same as -n
$ xtrdb noprompt       # backward compatibility (legacy, positional argument)
$ xtrdb -s data_file       # show the storage structure for the given file
$ xtrdb --storagemap file  # same as -s
$ xtrdb -h                 # help and build information, then exit
```

The `-n/--noprompt` mode removes highlighting, the `.` prompt, and the `ok` message — useful when input comes from a file or a pipe. The legacy positional variant `noprompt` still works too.

```
$ xtrdb -n < script.xtrdb
```

The `-s/--storagemap` option only runs the data-file structure report and exits the program (without entering the REPL).

Once started, the tool prints a `.` prompt and waits for a command. Every command ends with pressing Enter.

---

### Command overview

The `help` or `h` command shows the list of available commands:

```
$ 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 data
                                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
```

---

### Session management

| Command | Description |
|---|---|
| `exit`, `quit`, `q` | Exit the tool. Data not yet written to the database stays on disk. |
| `quitdrop`, `qd` | Exit and delete the open artifact files (data, `.desc`, `.meta`). |

---

### Environment configuration

| Command | Description |
|---|---|
| `storage [path]` | Set the working directory. Subsequent `open` commands look for the file at this path. |
| `policy [name]` | Set the storage policy (`DEFAULT`, `DIRECT`, `POSIX`, `MEMORY`, …). Must precede `open`. |

---

### Opening an artifact

```
open file_name
open file_name { TYPE field TYPE field ... }
```

If a `.desc` file exists, the schema is read from it. If not, the schema must be given in `{}`.

Array field types: `STRING name[8]` means a text field 8 bytes long (array multiplicity = 8).

Examples:

```
.open str1                          # schema from the file str1.desc
.open dump.tmp { INTEGER value }    # schema given manually
.open results { INTEGER a FLOAT b STRING name[8] }
```

---

**Reading and writing records**

| Command | Description |
|---|---|
| `read N` | Read record N (0-based) from the file into the payload buffer. |
| `rread N` | Like `read`, but reads from the end of the file (reverse read). |
| `write N` | Write the current payload to record N in the file. |
| `append` | Append the current payload as a new record at the end of the file. |
| `purge` | Delete all records from the file (truncate the file to 0 records). |

---

**Browsing content**

| Command | Description |
|---|---|
| `list N` | Print the first N records (from the beginning), one row per record. |
| `rlist N` | Like `list`, but reads from the end of the file. |
| `print` | Print the current payload in multi-line format. |
| `printt` | Print the current payload on a single line. |
| `size` | Print the record count and the size of a single record, in bytes. |
| `dump` | Print the raw bytes of the current payload in hex format. |
| `desc` | Print the field schema of the open artifact (multi-line). |
| `descc` | Print the schema on a single line (compact). |

---

**Editing the payload**

| Command | Description |
|---|---|
| `set field value` | Set the field with the given name in the payload buffer. |

| Command | Description |
|---|---|
| `setpos N value` | Set the field at index N (0-based) in the payload buffer. |
| `getpos N` | Print the value of the field at index N from the current payload. |
| `input` | Interactively fill the payload — enter values in order for each field. |
| `status` | Print the payload's state: `clean`, `fetched`, `changed`, `stored`. |
| `hex` / `dec` | Toggle numeric field input/output between hexadecimal and decimal. |

---

**Null metadata (`.meta`)**

| Command | Description |
|---|---|
| `meta` | Print the null and transmission-gap index from the `.meta` file — descriptively (segments with a record count and null pattern). |
| `metaraw` | Print the raw binary structure of the `.meta` file — every RLE entry with its `count`, `gap`, `bitsetHex` fields. |

`meta` shows segments with information about nulls and transmission gaps (`gap`). `metaraw` shows the raw binary structure of the `.meta` file.

---

**Other commands**

| Command | Description |
|---|---|
| `rox` | Toggle the "remove on exit" flag — deletes the data, `.desc`, and `.meta` when the tool exits. |
| `cap N` | Set the backread buffer capacity for stream devices. |
| `dropfile f1 f2 … }` | Delete the listed files. The list ends with the token `}`. |
| `echo text` | Print text to the terminal (useful in scripts). |
| `system command` | Invoke a shell command. |
| `#` or `rem` | A comment line (ignored). `#` doesn't even print a prompt. |

---

**Usage examples**
**Previewing an artifact**

```
$ xtrdb
.storage temp
.open str1
.size
.list 10
.quit
```

**Reading a DUMP file without a descriptor** Dump files created by `DO DUMP` have no `.desc` file — the schema must be specified manually:

```
$ xtrdb
.open results_alarm_dump.tmp { INTEGER value }
.size
.list 6
.quit
```

**Batch script**

```
xtrdb noprompt << 'EOF'
storage /var/retractor
open sensor_dump.tmp { INTEGER a FLOAT b }
list 20
quit
EOF
```

**Inspecting null metadata**

```
.open str1
.meta
.metaraw
```

## 8.2. System Origin

More than twenty years ago I worked at a scientific institute in Zabrze. Among other things, I was building a neonatal monitoring system. I had relatively recently finished my studies, and my head was still full of theory about building systems based on a central database. When building the monitoring system, I decided — I'll do it the way the discipline dictates — based on a relational database. That was not a good idea. I ran into a problem with the overall performance of such a solution. The recorded signals had high granularity. On top of that, the available database systems were not prepared for a continuous, unbounded stream of incoming data.

2003 was a time when so-called stream databases looked very promising in the scientific literature. After analysis, I decided that was probably the closest field at the time to what I needed. I adopted the assumption that I was building a streaming database for signal processing. Over time, that decision turned out not to be entirely accurate. Stream-processing systems came and went — but the need for systems processing time series remained. Stream-processing systems evolved into systems processing

time series — Time Series Databases. To this day, database systems that process time series are used in monitoring systems.

The neonatal monitoring system I developed served a dozen or so pulse oximeters. In the monitoring room lay a dozen or so newborns requiring continuous supervision. Each newborn was connected, among other things, to a pulse oximeter. Each pulse oximeter monitored heart rate and blood oxygen saturation. The newborns squirmed, probes fell off, and the pulse oximeters raised alarms every moment, reporting all sorts of problems. In such an information din, one of the newborns could be suffocating. It didn't happen suddenly — but slowly, and it could be recognized over a wider time horizon. That one case, however, required an immediate response. At the same time, several devices were signaling very loudly with sound — while the one newborn who needed help was quietly gasping for air in the corner of the room. That's roughly the scale of the problem. The system being built made it possible to tell, at a glance, whether a device wailing in the monitoring room was the result of a sensor slipping off, a momentary glitch, or something more serious. By changing the time scale, the problem could be identified immediately. A quick threat assessment based on the monitoring system's readings, in such a case, saves health and lives.

The monitoring system was built and deployed at a client, in one of Warsaw's hospitals. I was on site and saw it work. Unfortunately, inside it there was no data-management system of the kind I described in scientific publications. I built the solution by hand, without implementing a query language, algorithms, or management mechanisms. The deadline and limited resources required delivering the project on time. The publications that emerged at the time described noble needs and assumptions — but practice was different. A product had to be delivered, and there was no time.

This is, in broad outline, the generic reason that gave rise to the need to create a data-management system for signal processing. Over time, further application areas were added, arising from the expanding development areas related to telemetry, monitoring, and the growth of IoT systems.

## Why Was This Name Chosen for the System?

Retractors, in medicine, are an entire group of surgical instruments. Retractors are also known as surgical hooks or spreaders. These are tools that allow anatomical structures (e.g. wounds, muscles, bones, etc.) to be pulled apart or held together. They're typically used during surgery or another procedure. Some of the more inventive ones are named after their creators.

By analogy, I decided to name my tool a retractor. **RetractorDB**'s job is to separate, join, and enable real-time computation on time series, operating on the fly on ephemeral data, artifacts, or substrates (see the subchapter Artifacts, Substrates, Ephemerides).

**Info**

Definition (Data Retraction and Retractor): Using a numerical apparatus to extract, process, and then return data contained in time series or digital signals is called data retraction. The tool used to carry out this process is called a Data Retractor.

## 8.3. Further Development Directions

RetractorDB could potentially evolve into a more advanced form. Below I outline potential further development directions.

### Yet Another Mathematics

A few years ago I looked for a way to extend the algebra presented in the chapter on mathematical foundations to complex numbers. A direct application of Gaussian complex numbers — assuming the computational basis would be rational numbers — did not produce the expected results. The computational models showed that partitioning the set of natural numbers based on these numbers doesn't work.

My gut feeling was that the problem lay in the very nature of the complex numbers being processed. The modulus of a complex number whose both components are rational numbers is a Real number. Put differently, the length of the hypotenuse of a triangle whose legs are expressed as rational numbers lands in the set of real numbers.

The situation was uncomfortable. I started reviewing the literature. I came across something interesting — Eisenstein numbers. (Note — not Einstein, Eisenstein.) On Wikipedia you'll find an article titled "Eisenstein integers". Eisenstein — actually Ferdinand Gotthold Max Eisenstein — was a German mathematician who lived only 29 years, and left behind a contribution to mathematics that we can put to use.

Eisenstein integers are defined as:

$$z_C = a + b\omega \qquad a, b \in \mathbb{Z}$$

$$\omega = \frac{-1 + i\sqrt{3}}{2} = e^{\frac{2}{3}\pi i}$$

where the unit $i$ is the imaginary unit.

I decided to modify numbers presented this way as follows:

$$z_W = \frac{a}{b} + \frac{c}{d}\omega \qquad a, b, c, d \in \mathbb{Z}$$

And it's exactly these rational complex numbers that I used to build an algebra partitioning the set of natural numbers — and they work.

You can find the developed numerical model here: https://github.com/michalwidera/equations

## 8.4. RQL Syntax Highlighting

RetractorDB query files have the `.rql` extension. The repository ships ready-made syntax-highlighting definitions for three environments: Visual Studio Code, Vim, and the `bat`/`batcat` tool. All the necessary files live in the project's `scripts/` directory.

### Visual Studio Code

The `rql-vscode` extension adds full RQL language support to VS Code: syntax highlighting, recognition of the `.rql` extension, and a file icon.

**Installing from the GitHub repository:**

```
git clone https://github.com/michalwidera/rql-vscode.git
cd rql-vscode
npm install
npm run compile
code --install-extension *.vsix
```

If the repository contains a ready-made `.vsix` file, you can skip compiling and install it directly:

```
code --install-extension rql-vscode-*.vsix
```

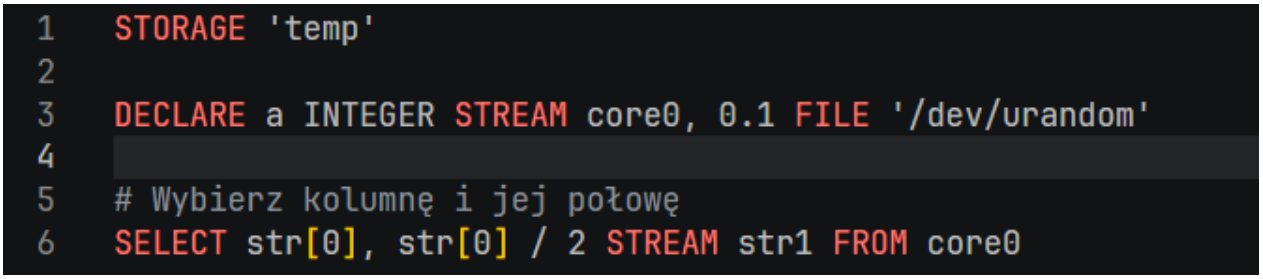
After installation, VS Code automatically recognizes `.rql` files and applies syntax highlighting. No user-settings changes are needed.

**Example of a highlighted query in VS Code:**

```
STORAGE 'temp'

DECLARE a INTEGER STREAM core0, 0.1 FILE '/dev/urandom'

## Select a column and half of it
SELECT str[0], str[0] / 2 STREAM str1 FROM core0
```




Highlighting - screenshot

*Fig. 62. RQL syntax highlighting in the Visual Studio Code editor*

As shown in Fig. 62, keywords (`STORAGE`, `DECLARE`, `SELECT`, `FROM`) are highlighted as commands, data types (`INTEGER`) as types, and comments starting with `#` or `//` as comments.

---

## Vim

The repository contains two Vim files in the `scripts/.vim/` directory:

| File | Description |
|---|---|
| `scripts/.vim/syntax/rql.vim` | Definition of the syntax groups and their color assignments |
| `scripts/.vim/ftdetect/rql.vim` | Automatic filetype detection by the `.rql` extension |

### Installing via buildrdb.sh

The most convenient method — the script copies both files into the appropriate `~/.vim/` subdirectories:

```
scripts/buildrdb.sh vimsyntax
```

The script creates any missing directories and reports the destination location:

```
-- RetractorQL vim syntax installed to /home/user/.vim
```

### Installing via CMake

The `vimconf` target from `scripts/CMakeLists.txt` copies the entire `.vim` directory into the home directory:

```
cmake --build build --target vimconf
```

### Manual installation

```
mkdir -p ~/.vim/syntax ~/.vim/ftdetect
cp scripts/.vim/syntax/rql.vim   ~/.vim/syntax/
cp scripts/.vim/ftdetect/rql.vim ~/.vim/ftdetect/
```

After installation, Vim automatically activates highlighting for every file with the `.rql` extension. The file `ftdetect/rql.vim` contains a single line:

```
au BufRead,BufNewFile *.rql set filetype=rql
```

### Highlighted elements

| Vim group | Examples |
|---|---|
| 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 | # comment, // comment, /* block */ |
| String | 'path/to/file.dat' |
| Number | 42, 3.14, 1/2, 1e5 |

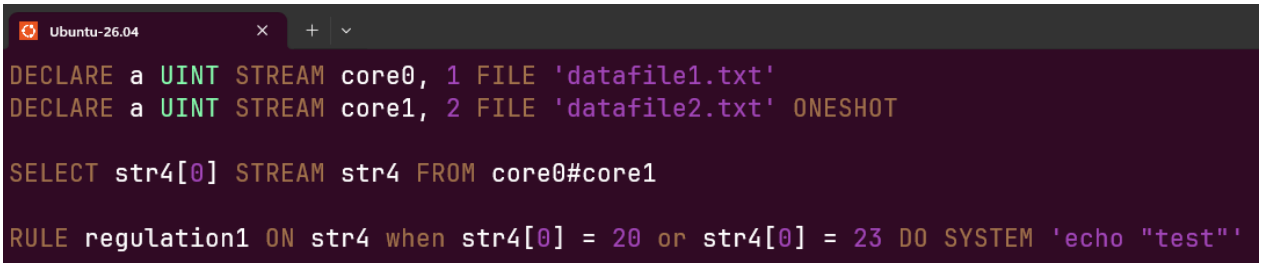
**Example query file with highlighted fragments:**

```
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"'
```

The text view in the vim editor is shown in Fig. 63.




view in the vim editor

*Fig. 63. RQL syntax highlighting in the vim editor*

---

## bat / batcat

The `bat` tool (available as `batcat` on some distributions) is an improved `cat` replacement with built-in syntax-highlighting support. It supports Sublime Text 3 format syntax definitions, which the RetractorDB repository provides at `scripts/sublime/retractorql.sublime-syntax`.

### Prerequisite

Make sure `bat` is installed:

```
## Debian/Ubuntu
sudo apt-get install bat

## Check which command is available (may be bat or batcat depending on the distro)
command -v batcat || command -v bat
```

### Installing via buildrdb.sh

```
scripts/buildrdb.sh batsyntax
```

The script automatically detects the command (`bat` or `batcat`), copies the syntax file into the correct config directory, and rebuilds the syntax cache:

```
-- RetractorQL syntax installed to /home/user/.config/bat/syntaxes
```

### Manual installation

```
## Detect the command name
BAT=$(command -v batcat || command -v bat)

## Create the directory for syntax definitions
mkdir -p "$($BAT --config-dir)/syntaxes"

## Copy the definition
cp scripts/sublime/retractorql.sublime-syntax "$($BAT --config-dir)/syntaxes/"

## Rebuild the cache
$BAT cache --build
```

### Usage

After installation, `bat` automatically highlights `.rql` files:

```
bat query.rql
```

The `.desc` extension (stream descriptor files) is also recognized. Highlighting can be forced manually if the file has a different extension:

```
bat --language rql any-file.txt
```

### Verifying the installation — available languages:

```
bat --list-languages | grep -i rql
## RetractorQL:rql,desc
```

**Example invocation**

For a file `query.rql` containing:

```
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"'
```

Running `bat query.rql` displays the file's content with line numbers and syntax highlighting in the terminal, where keywords, types, comments, and string literals get distinct colors according to `bat`'s active theme (Fig. 64).

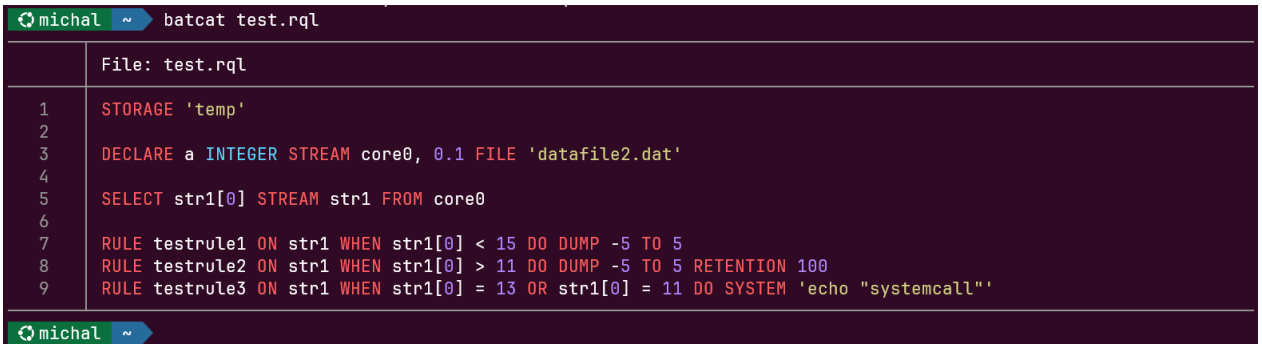



Fig. 64. RQL syntax highlighting in the terminal — the batcat command

## 8.5. Integration Tests

Integration tests verify the system's behavior as a whole — they run the actual binaries (`xretractor`, `xqry`, `xtrdb`) and compare their output against patterns, or check specific properties of the output files. This differs from unit tests, which use the GTest framework to test isolated classes and functions of the `rdb` and `retractor` libraries (e.g. `payload`, `descriptor`, `crsMath`, `compiler`), don't require a running server, and don't produce artifacts on disk. Integration tests are run with the command `ninja test` (or `ctest`) in the `build/Debug/` directory; a single test can be run with `ctest -R <name> -V`.

Integration tests are split into two directories based on their concurrency requirements. Tests in the `IntegrationTest_serial` directory run the `xretractor` server in IPC mode — they use the shared lock file `/tmp/xretractor_service.lock`

and Boost shared-memory segments. To avoid conflicts between concurrent instances, CMake forces `RUN_SERIAL TRUE` for them (one after another). Tests in the `IntegrationTest_parallel` directory do not start the IPC server — they compile queries (`xretractor -c`) or perform file operations via `xtrdb` — and can safely run in parallel.

### Serial tests — `IntegrationTest_serial`

| Test name | Description |
|---|---|
| `agse1` | The `@(start, length)` time-window operator — forward variants `@(1,4)`, backward `@(1,-4)`, various lengths. See: *AGSE Sliding Data Window*. |
| `agse2` | Window `@(n,m)` combinations on a 3-field stream, alignment and rate conversion at ratios 1:1, 1:2, 2:3, 2:4. See: *AGSE Sliding Data Window*. |
| `agse3` | The `@(n,m)` operator when the output rate is lower than the input rate (source rate 0.1) — windows `@(3,2)`, `@(3,3)`, `@(3,-3)`. See: *AGSE Sliding Data Window*. |
| `consistency` | Read consistency: two streams read the same source; their difference must always equal 100. See: *Data and Control Flow*. |
| `issue113_meta_internal` | Structure of the `.meta` sidecar file: header size (8 B), entry size (18 B), sampling interval, null bitsets for records with and without nulls. See: *Data Storage Format — Files*. |
| `issue113_meta_xtrdb` | Verification via `xtrdb` that after running `xretractor+xqry`, the `.meta` file is created and reported correctly (`meta: temp/str_null.meta`). See: *Data Storage Format — Artifact Analysis*. |
| `issue113_null_skip` | The `-n` flag in `xqry` — rows that are entirely null are skipped; without the flag, all rows (including all-null ones) must be present. See: *Command-Line Options — xqry*. |
| `issue113_null_xqry` | Nulls transmitted over IPC: null values from the source file are shown as `null` in `xqry` output. See: *Command-Line Options — xqry*. |

| Test name | Description |
|---|---|
| `issue121_isnull` | The `isnull(field)` function — returns 1 when the field is null, 0 otherwise. See: *Aggregate Operators and Expression Functions*. |
| `issue121_null_propagation` | Propagation of null values through `SELECT` to the output stream. See: *SELECT Command*. |
| `issue128_numeric_to_string` | Conversion of `INTEGER`/`FLOAT` to `STRING` via the `to_string()` function with a declared field width; verification of the output descriptor. See: *Aggregate Operators and Expression Functions*. |
| `issue128_string_to_numeric` | Conversion of `STRING` to numeric types: `to_integer()`, `to_float()`, `to_double()`; null propagation through the conversion. See: *Aggregate Operators and Expression Functions*. |
| `issue167_dedup_cascaded` | Cascaded absorption of substrates via `deduplicateSubstrats()` — multi-step rewriting of `PUSH_ID` tokens. See: *Substrates*. |
| `issue167_dedup_field_names` | Substrate deduplication without comparing schema field names — merging when field types are equivalent, regardless of names. See: *Substrates*. |
| `issue167_dedup_nonzero_offset` | Updating `PUSH_ID` in the consumer's `lSchema` for a non-zero offset of the absorbed substrate; coverage of the code path in `compiler.cpp`. See: *Substrates*. |
| `issue167_dedup_positive` | The basic deduplication case: a substrate merged with a named stream with an equivalent program and field types. See: *Substrates*. |
| `issue167_triarg` | Multi-argument stream expressions: `s1+s2+s3`, `(s1#s2)#s3`, `s1+(s2#s3)`, `s1+s2+s3+s4`; in-memory and on-disk substrates. See: *Substrates, Operation Sequencing*. |
| `issue42_rule` | The `RULE` command — conditional `DUMP` and `SYSTEM` actions triggered on stream values; runs `xretractor` and reads the result via `xtrdb`. See: *RULE Command*. |

| Test name | Description |
|---|---|
| `issue56_timeshift` | The `>` filter operator on joined streams — only records satisfying the condition end up in the output stream. See: *SELECT Command — Sequencing*. |
| `issue61_tmpmem` | The in-memory substrate `SUBSTRAT 'memory'` — intermediate data kept in RAM instead of on disk. See: *STORAGE Types*. |
| `issue6_adhoc` | Ad-hoc query mode: `xqry -a 'SELECT ...'` — defining and executing a query on the fly, without an `.rql` file. See: *Ad Hoc Queries*. |
| `operations` | The `#` (HASH merge) operator on two streams with different rates — verifying the ratio of record counts in the output. See: *Interleaving Operation Sequencing*. |
| `rotation_test` | The stream binary-file rotation mechanism (`ROTATION`) — file count after two `xretractor -m 2` cycles. See: *File Rotation Mechanism*. |
| `simple` | A smoke test for arithmetic on joined streams (`core0` rate 0.1 + `core1` rate 0.2), read via `xtrdb`. See: *SELECT Command*. |
| `simple_max` | The `.max` operator on a stream — the maximum value, joined with the original stream. See: *Aggregate Operators and Expression Functions*. |
| `xqry_elem_limit` | The `-m N` parameter in `xqry` — limits the number of received records to exactly N, regardless of the source's length. See: *Command-Line Options — xqry*. |

### Parallel tests — `IntegrationTest_parallel`

| Test name | Description |
|---|---|
| `dsp` | Regression for the FIR filter pipeline: sliding window `@(1,25)`, array multiplication with the `_` index, `.sumc` reduction, joining the signal with the output. See: *Signal Filter Implementation*. |

| Test name | Description |
|---|---|
| `issue113_meta` | `xtrdb` operations after two `appends` — record list and hexdump of the binary file compared against a pattern. See: *Data Storage Format — Artifact Analysis*. |
| `issue113_meta_autocreate` | Automatic creation of the `.meta` sidecar file after the first `append`; size >16 B; `xtrdb` reports the correct path. See: *Data Storage Format — Files*. |
| `issue113_null_txtsrc` | The `rread`/`getpos` commands in `xtrdb` on a TEXTSOURCE stream containing nulls. See: *Data Storage Format — Artifact Analysis*. |
| `issue153_storagemap_meta_cases` | The `xtrdb -s` storage map for a plain file and a retractordb-style file: slot markers, segment list, rotated files, `.meta`/`.shadow` references. See: *Inspection Tool xtrdb -s*. |
| `issue31_doc` | Generating DOT/SVG graphs via `xretractor -c -d ...` for documentation examples at three levels of detail. See: *Compilation Debugging*. |
| `issue42_rule` | Compilation of `RULE` syntax — the `-c` stage only, without starting the server. See: *RULE Command*. |
| `issue56_timeshift` | Compilation of the `>` filter operator — the `-c` stage only. See: *SELECT Command — Sequencing*. |
| `issue61_tmpmem` | Compilation of a query with `SUBSTRAT 'memory'` — the `-c` stage only. See: *STORAGE Types*. |
| `issue95_loopInCompile` | Cycle detection in the query graph by the compiler — expected “Circular dependency” error and a non-zero exit code. See: *Loop Detection in Compilation*. |
| `issue96_no_substrat_reduction` | User-defined streams with an identical structure are NOT merged; only automatic substrates are subject to merging. See: *Substrates*. |
| `issue96_substrat_reference` | A generated substrate shared by two user streams — correct references in the dependency tree. See: *Substrates*. |

| Test name | Description |
|---|---|
| `Pattern1` | Compilation of the `#` (HASH-merge) operator, field selection with an offset, and stream joining with `+`. See: *Interleaving and Summation Operation Sequencing*. |
| `Pattern2` | Compilation of queries on `BYTE` streams from `/dev/urandom`: `SELECT`, arithmetic, stream joining. See: *SELECT Command*. |
| `Pattern3` | Compilation of `SELECT *` (unfold) with an output-file declaration and retention. See: *Asterisk Expansion*. |
| `Pattern4` | Compilation of the `Crc(bits, seed)` function in 16-bit and 8-bit variants on a 2-field stream. See: *Aggregate Operators and Expression Functions*. |
| `Pattern5` | `xtrdb` operations on a multi-type record (STRING, INTEGER, BYTE, FLOAT): `append`, `read`, `list/rlist`, `input`, `write`. See: *Artifact Analysis*. |
| `Pattern6` | Compilation of the window operator `@(n,m)` forward `@(1,10)` and backward `@(1,-10)`, plus a leak-free valgrind run. See: *AGSE Sliding Data Window*. |
| `Pattern7` | Compilation with identical field names across multiple streams (issue #17) — correct field identification via the stream index. See: *DECLARE Command, Aliasing*. |
| `retention` | `xtrdb` operations on a file with a `RETENTION` parameter: `open`, `purge`, `append`, `list`, `write` for a specific record. See: *Data Storage Format — File Rotation Mechanism*. |
| `simple` | Compilation of a basic arithmetic query + DOT graph + valgrind; uses data from `IntegrationTest_serial/simple`. See: *SELECT Command*. |
| `simple_max` | Compilation of a query with `.max` + DOT graph + valgrind; uses data from `IntegrationTest_serial/simple_max`. See: *Aggregate Operators and Expression Functions*. |

| Test name | Description |
|---|---|
| `subquery` | |
| | Compilation of nested subqueries: `(a#b)>1` (hash-merge inside a filter) and `(a>1)#b` (filter inside a hash-merge). See: *Dependency Tree Construction*. |
| `txtsrc` | |
| | `xtrdb` operations `descc`/`rread`/`printt` on a TEXTSOURCE stream (a text file as a data source). See: *Data Storage Format*. |

# Chapter 9

# References


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, vol. 21, pp. 6-27, 1969. (link)

3. M. Widera, “Deterministic method of data sequence processing” Annales UMCS, Sectio AI Informatica, vol. IV, pp. 314-331, 2006 (link); Polish version: “Deterministyczna metoda przetwarzania ciagow danych” in XXI Autumn Meeting of Polish Information Processing Society, Conference Proceedings, pp. 243-254, 2006. (pdf)

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. [Accessed: 3 Nov 2025]. (link)

7. A. Staltz, “RxJS Marbles” https://github.com/staltz/rxmarbles, [Online]. Available: https://rxmarbles.com/. [Accessed: 4 Nov 2025]. (link)

8. “Conan.io - the Open Source C and C++ Package Manager for Developers” JFrog, [Online]. Available: https://conan.io/. [Accessed: 9 Nov 2025].

9. D. W. Gunness, “Creating digital signal processing (DSP) filters to improve loudspeaker transient response”. US Patent US8081766B2, 20 Dec 2011. (link)

10. M. Widera, “RetractorDB - separator serii czasowych” Programista, vol. 92, no. 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, vol. 64, no. 2, pp. 273-298, 1997.

12. L. Schaeffer, J. Shallit and S. Zorcic, “Beatty Sequences for a Quadratic Irrational: Decidability and Applications” arXiv:2402.08331, 2024. (pdf)

13. M. A. Berger, A. Felzenbaum and A. S. Fraenkel, “Disjoint covering systems of rational Beatty sequences” Journal of Combinatorial Theory, Series A, vol. 42, no. 1,

pp. 150-153, 1986.

14. D. Eppstein et al., “Aperiodic pinwheel scheduling using Beatty sequences” — a discussion of the periodic-scheduling problem based on complementary Beatty sequences, 2023. (link)

15. H. Fujiwara, K. Miyagi and K. Ouchi, “Pinwheel Scheduling with Real Periods” arXiv:2510.24068, 2025 — proofs based on the Rayleigh/Beatty partition, with identities on floor and ceiling functions. (html)

16. S. Samadi, M. O. Ahmad and M. N. S. Swamy, “Characterization of nonuniform perfect-reconstruction filterbanks using unit-step signal” IEEE Transactions on Signal Processing, vol. 52, no. 9, pp. 2490-2499, 2004. (link)

17. G. Margolis and Y. C. Eldar, “Nonuniform Sampling of Periodic Bandlimited Signals” IEEE Transactions on Signal Processing, vol. 56, no. 7, pp. 2728-2745, 2008. (pdf)

18. J. Kovačević and M. Vetterli, “Perfect Reconstruction Filter Banks with Rational Sampling Factors” IEEE Transactions on Signal Processing, vol. 41, no. 6, pp. 2047-2066, 1993.

19. S. Kalra and N. K. Shukla, “Ramanujan sums in signal recovery and uncertainty principle inequalities” arXiv:2512.16190, 2025. (pdf)

20. A. Arasu, S. Babu and J. Widom, “The CQL continuous query language: semantic foundations and query execution” The VLDB Journal, vol. 15, no. 2, pp. 121-142, 2006. (pdf)

21. J. Krämer and B. Seeger, “Semantics and implementation of continuous sliding window queries over data streams” ACM Transactions on Database Systems, vol. 34, no. 1, pp. 1-49, 2009 (the PIPES system).

22. S. K. Jensen, T. B. Pedersen and C. Thomsen, “Time Series Management Systems: A Survey” IEEE Transactions on Knowledge and Data Engineering, vol. 29, no. 11, pp. 2581-2600, 2017. (pdf)

23. K. O’Bryant, “Fraenkel’s partition and Brown’s decomposition” Integers: Electronic Journal of Combinatorial Number Theory, vol. 3, A11, 2003.

24. G. E. Pfander, S. Revay and D. Walnut, “Exponential bases for partitions of intervals” Applied and Computational Harmonic Analysis, vol. 68, art. 101607, 2024.

25. M. Widera, J. Jezewski, R. Winiarczyk, J. Wrobel, K. Horoba and A. Gacek, “Data stream processing in fetal monitoring system: I. Algebra and query language” Journal of Medical Informatics & Technologies, vol. 5, pp. 83-90, 2003.

# RetractorDB

Edge Signal Processing Engine

Michal Widera

2026-07-12 · commit a083c6d