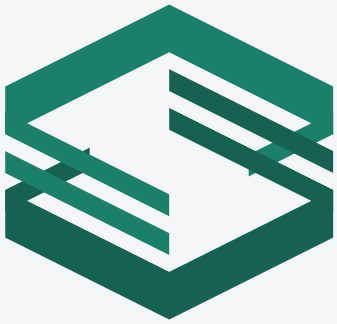
Manual 09 · measured numbers, sizing, good habits

Performance & Best Practices



What the engine achieves on measured benchmarks, why it behaves the way it does, how to size a run machine, and the practices that keep flows fast, correct and easy to operate — each with an example.

CONTENTS

01 **Measured performance**
- 1.1 Benchmarks
- 1.2 Start-up
- 1.3 How it compares

02 **Why it is fast, and what that means for you**
- 2.1 Sizing a run machine

03 **Best practices for speed**
- 3.1 Let the database do the work
- 3.2 Keep in memory only what is small
- 3.3 Write in batches, commit once
- 3.4 Keep logs light
- 3.5 Parallelise with lanes

04 **Best practices for correctness**
- 4.1 Make failures loud and complete
- 4.2 Declare what the data must respect
- 4.3 Test like code

05 **Best practices for operations**
- 5.1 A checklist before going to production

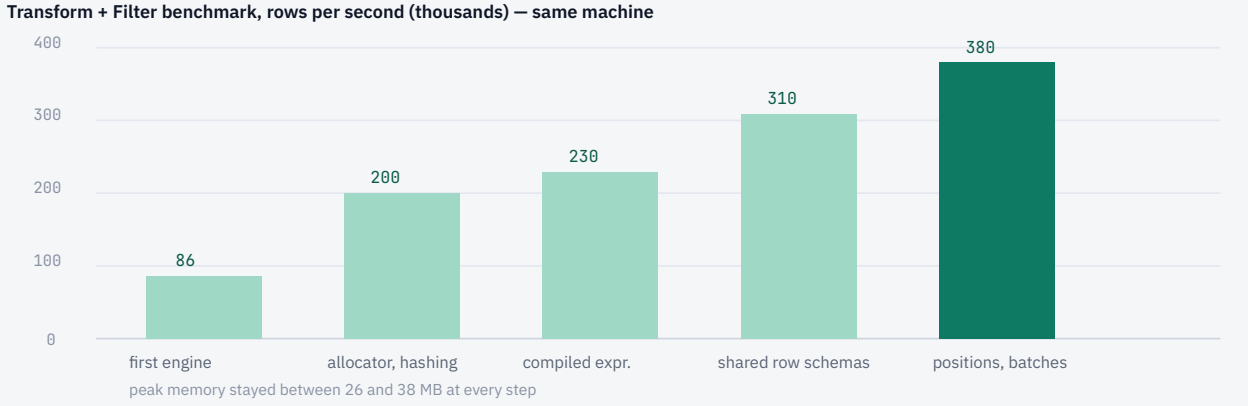
Copyright © 2025–2026 Marco Dongu. All rights reserved.

01 Measured performance

All numbers in this chapter come from the engine's own benchmarks, run with the same nodes and the same compiled plan the studio produces. They were measured on a small virtual machine — **2 vCPU Xeon at 2.1 GHz** — so absolute values on a real server will be higher; what matters is the order of magnitude and the memory profile.

Benchmarks

Benchmark	Rows	Throughput	Peak memory
CSV (5 columns, 35 MB) → Transform (3 FPEL expressions) → Filter → CSV	1 000 000	**~380 000 rows/s** (2.6 s)	**33 MB**
the same, with field mapping on input and output	1 000 000	~356 000 rows/s	33 MB
TMap: 500 000 orders × 9 000 customers, lookup join, 2 filtered outputs	500 000	~273 000 rows/s	—
Script: `let`, `if`, assignments, `upper`	1 000 000	+14% over the previous engine version	—



Start-up

Step	Time
Admission (`--admit-only`: signature, policy, configuration)	**3 ms**
Complete run of a minimal plan, process start to exit	**8–17 ms**

A runner that starts in milliseconds can be launched by a scheduler every minute, or once per incoming file, without a resident service.

How it compares

We have not benchmarked other products side by side, so this is a description of design differences, not a measured comparison:

	Stilton runner	Typical JVM-based ETL runtime
Start-up	milliseconds; no virtual machine to warm up	seconds, before the first row
Memory for a streaming flow	tens of MB, flat with the input size	hundreds of MB of heap reserved up front
Pauses	none: no garbage collector	collection pauses, tunable
Distribution	one executable, no runtime to install	a JVM of the right version on every machine
Throughput on I/O-bound jobs	limited by the database, network or disk — as for any engine	same

Note. For most real jobs the language of the engine is not the bottleneck: the database, the network and the disk are. Measure your own flows with the monitor panel: the *Nodes — timing and throughput* section shows where the time goes.

02 Why it is fast, and what that means for you



- **Rows stream** through bounded channels in small batches. A slow consumer makes the producer wait instead of filling memory.
- **Expressions are compiled once** per node and evaluated without lookups by name; each node remembers where a field sits in the row.
- **Rows share their schema**: field names are stored once, not in every row.
- **Lanes run in parallel** on all CPU cores; inside a lane every node is an independent task.

Sizing a run machine

Flow	Memory to plan
only streaming nodes	50–100 MB, whatever the volume
with dataset nodes	the rows they hold × the size of a row in memory (roughly 2–3 × its size in CSV) + 100 MB

Flow	Memory to plan
several lanes with dataset nodes	the sum of the lanes that hold data at the same time

Example. An Aggregate over 5 million rows of 200 bytes each (1 GB as CSV) holds about 2–3 GB while it reads. If the source is a database, let the database aggregate instead (`GROUP BY` in the query) and the engine reads a few thousand rows.

CPU: one core per lane that works at the same time is a good start; a lane with a TMap or a Script benefits from a second core for its outputs.

---

03 Best practices for speed

Let the database do the work

The fastest row is the one never read.

Instead of…	Do
DB Input of a whole table → Filter	keep the Filter right after the DB Input: its conditions are pushed into the query automatically (check the log: `pushed into the query: WHERE …`)
DB Input → Aggregate	write the `GROUP BY` in the query
DB Input → Join with another table of the same database	write the `JOIN` in the query
DB Input with `SELECT *`	select only the columns you use

Keep in memory only what is small

- In **Join** and **TMap**, the **lookup** side is held in memory: make it the smaller side. 500 000 orders against 9 000 customers — customers are the lookup.
- A **Materialize** read by several nodes is stored once; three Joins on the same reference table should share one Materialize instead of reading the table three times.
- The **JSON/XML parsers** work one incoming document at a time; **Aggregate**, **Window**, **Pivot** and **Explode** from a dataset hold everything — filter before them, not after.

Write in batches, commit once

- **DB Sink**: a batch size of 500–5 000 rows; a native transaction group commits once at the end of the lane instead of after every batch.
- **TRUNC+INS** on a staging table, then a post-write SQL `INSERT … SELECT` into the final table, is often faster than UPSERT row by row.
- **File Output** with partitions writes many files in one pass; up to 1 000 partitions open at once.

Keep logs light

- Use the **Normal** or **Essential** log level for artifacts. *Diagnostic* sends every row to the monitor and slows everything down.
- A **Log** node on a large flow writes every row: sample it (*one every N rows*) or disable it in production.

Parallelise with lanes

Two independent loads in one lane run as one chain; in two lanes they run in parallel on two cores. Split a flow into lanes where it naturally splits, and join them with bridges only where data must cross.

04 Best practices for correctness

Make failures loud and complete

- Mark as **critical** the nodes without which the lane must not continue — typically the sources.
- Connect every `reject` port to something: a file, a table. A rejected row that goes nowhere is lost silently.
- Connect each lane's `error_out` to a notification (Mail) or a table.
- Use **transactions** for writes that belong together (header and lines), so a failure leaves nothing half-written.
- Before writing to a shared target, consider a **Bridge In** in *All at once* mode: the target receives everything or nothing.

Declare what the data must respect

`expect` rules in a Script document the assumptions and count their violations in every run:

Script — text

```
expect order_id is not null                 else error  "order without id"
expect amount ⩾ 0                           else reject "negative amount"
expect order_date ⩽ today()                 else warn   "order in the future"
expect country in ["IT", "FR", "DE", "ES"]  else warn   "unexpected country"
```

The monitor shows, run after run, how many rows broke each rule: a sudden change is often the first sign that a source changed.

Test like code

- Keep a few **example inputs** with **expected outputs** next to each project, and run them after every change — the `examples` folder of the beta shows the pattern, with a verifier that compares the run's statistics with the expected ones.
- Write **examples** for every user function: they run at every Run and Compile.
- Use **Preview** and **explain this value** while designing; use the monitor panel's timeline before deploying.

05 Best practices for operations

Practice	Why
Clear names for nodes, lanes and artifacts (`Read orders`, `orders-daily-prod.ffart`)	logs, the monitor and the people on call read names
Every environment-specific value is a `${VARIABLE}`, every password a `${SECRET}`	one artifact per environment, no secret in files
A named checkpoint in History (or a Git tag) for every artifact you deploy	you can always tell what is running
`--admit-only` before scheduling a new artifact	refusals are found at deploy time, not at 3 a.m.
Runtime profile on production machines, denying Shell and SSH unless needed	a signed artifact cannot do more than the machine allows
The monitor's ingest token on runners, its read token only to operators	each token can do one thing
Keep the complete run records (`~/.stilton/runs`) for as long as your audit requires	the monitor may hold only the essentials

A checklist before going to production

- [ ] the flow ran in the studio on realistic volumes; the monitor panel shows no node far slower than the others
- [ ] memory of the dataset nodes checked against the run machine
- [ ] critical nodes marked; rejects and `error_out` connected
- [ ] `expect` rules on the assumptions that matter
- [ ] variables and secrets instead of literal values; profile frozen, configurable values decided
- [ ] artifact compiled with log level *Normal* or *Essential*, monitor set
- [ ] `--admit-only` passes on the target machine; required secrets present
- [ ] schedule or service in place, with the right account and working directory
- [ ] someone knows where the monitor is and who receives the error mails