Filter Representation in Vectorized Query Execution (DAMON’21) — Junior-Friendly Explanation

Paper: **“Filter Representation in Vectorized Query Execution”** (Ngom, Menon, Butrovich, Ma, Lim, Mowry, Pavlo), 2021.

This write■up explains the paper in a way that a junior data engineer can follow. The goal is to understand:

- what “vectorized execution” means
- what “selection vectors” and “bitmaps” are
- why one is sometimes faster than the other
- what the paper’s recommendations are

---

1) The problem (in plain language)

Modern analytical databases often keep data in **memory** (RAM).

That changes the bottleneck:

- Old world: disk was slow → optimize disk reads
- New world: memory is fast → CPU efficiency matters a lot

A popular CPU efficiency trick is **vectorized execution**.

What is vectorized execution?

Instead of processing one row at a time, the engine processes a **batch** (a “vector”) of rows at a time. Typical batch size: ~1–2k tuples.

Why this is faster:

- less function-call / iterator overhead
- better cache usage
- more chances to use **SIMD** (CPU instructions that do the same operation on multiple values at once)

---

2) Where “filters” come in

During query execution, operators keep narrowing the data:

- scan a table
- apply WHERE conditions
- compute expressions
- joins, groups, etc.

After each step, the system must track:

> which rows in the current batch are still “alive” (valid)

The paper calls that tracking structure a **filter representation**.

---

3) Two filter representations

A) Selection Vector (SV)

A **selection vector** is basically a list of indexes (tuple IDs) that survived.

Example:

- Batch has 8 rows: positions `0..7`
- Rows that survived: `0, 3, 4, 7`
- Selection vector: `[0, 3, 4, 7]`

Mental model:

- It’s like a guest list: only these people are allowed in.

Cost shape:

- Great when few rows survive (small list).
- But it introduces “indirection”: you read `idx = SV[i]` then access `col[idx]`.

B) Bitmap (BM)

A **bitmap** stores 1 bit per row in the batch.

Example:

- Batch has 8 rows
- Survived rows `0, 3, 4, 7`
- Bitmap: `1 0 0 1 1 0 0 1`

Mental model:

- It's like a row of light switches; ON means "row is valid."

Cost shape:

- Great when many rows survive (dense selection).
- Also great when you can process whole vectors with SIMD.
- But iterating "only the 1s" can be expensive (bit scanning logic).

---

4) Two types of operations: Update vs Map

The paper distinguishes two primitives over filtered vectors:

Update

- Applies a predicate and **changes** which rows are selected.
- Example: WHERE clause filtering rows.

Map

- Computes a new vector but **does not change** which rows are selected.
- Example: projection like `SELECT price * 1.2`.

Why this distinction matters:

- Some "full compute" strategies are only valid for some combinations.

---

5) Two compute strategies: Selective vs Full

Given a filter representation, you still have a choice:

Selective compute

Only process rows that are selected.

- If selectivity is low (few rows survive), this saves work.

Full compute

Process all rows in the batch regardless of selection.

- Sounds wasteful, right?
- But it can be faster if the operation is SIMD-friendly and the iteration is cheaper.

This is one of the paper's key messages:

> Sometimes doing "more work" is faster because the work is simpler and vectorizes well.

---

6) Why the "best" choice depends on selectivity

Selectivity = fraction of rows that are selected.

- 0.05 selectivity means 5% of rows survive
- 0.80 selectivity means 80% of rows survive

Think of it like a school bus:

- If only 3 kids are riding, you might drive directly to those 3 houses (selective).
- If 90 kids are riding, you just drive the full route (full), because skipping streets is harder than it's worth.

---

7) The paper's main finding (the "rule of thumb")

From their microbenchmarks and analysis:

- **Bitmaps tend to perform better** when the operation can be efficiently **SIMD vectorized**.
- **Selection vectors tend to perform better** for other cases because iteration logic is cheaper.

More detailed:

- SVs are great when selectivity is low (few rows survive) and you want to iterate only survivors cheaply.
- BMs are great when you can use full scans + SIMD, especially at higher selectivities.

---

8) What the paper actually recommends (practical engineering)

The authors argue that a good vectorized engine should:

1) Support **both** representations (SV and BM) 2) Pick representation/strategy dynamically based on:

- operation cost

- iteration cost

- SIMD friendliness

- selectivity

They show that “one strategy always” is not optimal.

---

9) Why this matters to you (junior data engineer angle)

Even if you never implement a DBMS, the ideas map to real engineering:

- **Dense vs sparse data structures**
- **Branchy code vs vectorized code**
- **Doing less work vs doing simpler work**

This shows up in:

- analytics engines

- columnar formats
- SIMD-friendly transformations
- vectorized UDF execution

If you work on performance-sensitive pipelines, you'll keep seeing this pattern:

> The fastest plan is the one that matches the hardware.

---

10) Quick glossary

- **Tuple**: a row.
- **Vector / batch**: a small group of rows processed together.
- **Selectivity**: fraction of rows that pass a filter.
- **SIMD**: CPU instructions that operate on multiple values per instruction.
- **Selection vector**: list of surviving row indexes.
- **Bitmap**: bitmask of surviving rows.

---

11) Takeaways (memory-friendly)

If you only remember 3 things:

1) Vectorized engines process batches to reduce overhead and use caches better. 2) After each operator, you must track which rows are still valid. 3) SV vs BM is a tradeoff; the best depends on selectivity + SIMD ability.