DB 2

08 – Predicate Evaluation

Summer 2022

Torsten Grust
Universität Tübingen, Germany

1 ┆ Q_7 — Predicate (or Filter) Evaluation

SQL's WHERE/HAVING/FILTER clauses use **expressions of type Boolean** (**predicates**) to filter rows. Predicates may use Boolean connectives (AND, OR, NOT) to build complex filters from simple predicate building blocks:

```
SELECT t.a, t.b
FROM   ternary AS t
WHERE  t.a % 2 = 0 AND [OR] t.c < 1  -- either AND or OR
```

Evaluate predicate for every row t scanned. Here: assume that evaluation of the predicate is *not* supported by a specific index. (⚠️ Index support for predicates is essential → see upcoming chapters.)

Using **EXPLAIN** on Q_7

```
EXPLAIN ANALYZE VERBOSE
  SELECT t.a, t.b
  FROM   ternary AS t                 -- 1000 rows
  WHERE  t.a % 2 = 0 AND t.c < 1;
```

```
                                   QUERY PLAN

                                          ↓                          ↓
 Seq Scan on ternary t (cost=… rows=1 …)  (actual time=… rows=4 …)
   Filter: ((c < '1'::double precision) AND ((a % 2) = 0)) ←
   Rows Removed by Filter: 996
 Planning time: 2.125 ms      ↑
 Execution time: 1.894 ms
```

- Filter predicate evaluated during Seq Scan.
- Estimated **selectivity** of predicate $^1/_{1000}$ (real: $^4/_{1000}$).

t.a % 2 = 0 AND t.c < 1: An Expression of Type **bool**

- In the absence of index support, use the regular expression interpreter to evaluate predicates:

```
  SCAN_FETCHSOME(t, [a, c])
  SCAN_VAR(c)
  CONST(1)
? FUNCEXPR_STRICT(<, •, •)
  BOOL_AND_STEP_FIRST(      •)      # if • = false, immediately yield false
  SCAN_VAR(a)                       #      (∧ semantics: false ∧ p = false)
  CONST(2)
  FUNCEXPR_STRICT(%, •, •)
  CONST(0)
  FUNCEXPR_STRICT(=, •,     •)
  BOOL_AND_STEP_LAST(           •)  # yield •   (∧ semantics: true ∧ p = p)
```

- Uses jumps (⇣) in program to implement **Boolean shortcut.**

Heuristic Predicate Simplification

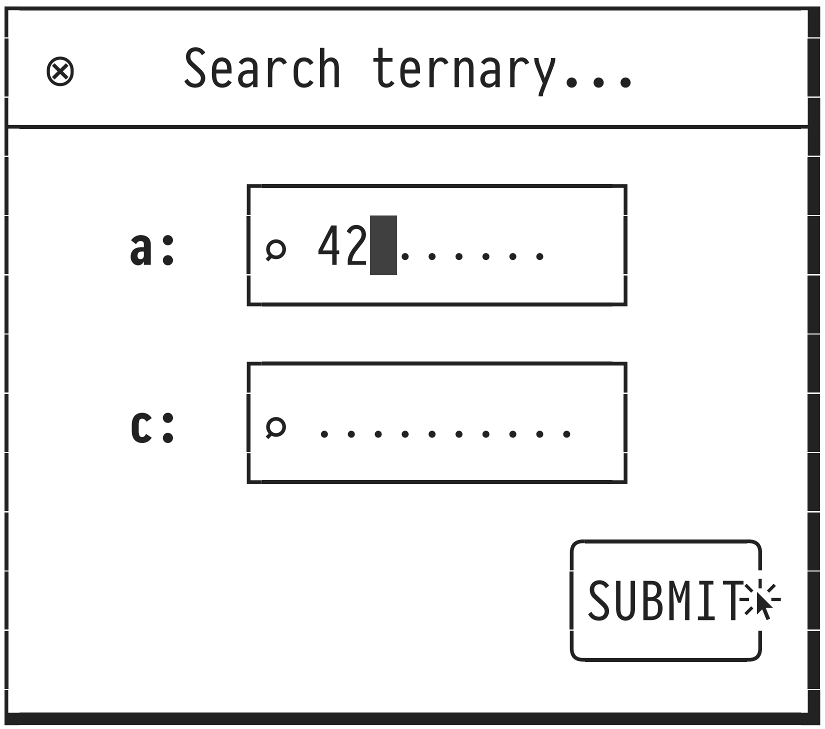
- Predicate evaluation effort is multiplied by the number of rows processed. **Even small simplifcations add up.**

- PostgreSQL performs basic predicate simplifications:
  - Reduce constant expressions to true/false.
  - Apply basic identities (e.g., NOT(NOT(p)) ≡ p and (p AND q) OR (p AND r) ≡ p AND (q OR r)).
  - Remove duplicate clauses (e.g., p AND p ≡ p)
  - Apply De Morgan's laws.

- ⚠️ These are **heuristics** (expected to improve evaluation time): selectivity is *not yet* taken into account.

Machine-Generated Queries and Predicate Simplification

Automatically generated SQL text may differ significantly from human-authored queries. Consider a web search form:



1. User enters search keys for columns **a** and/or **c**.

2. Web form maps missing keys to **NULL** (interpret as wildcard).

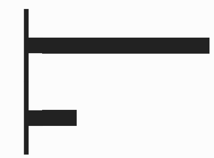
3. DBMS executes parameterized query:

```
SELECT t.*
FROM   ternary AS t
WHERE  (t.a = :a OR :a IS NULL)
  AND  (t.c = :c OR :c IS NULL)
```

Heuristics May Not Be Enough

- Heuristics only go so far. The (estimated) **cost** of evaluation may suggest better predicate rewrites:

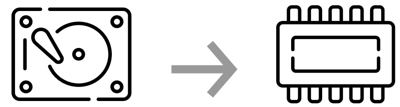
```
SELECT t.*                                            (expected) cost
FROM   ternary_10m AS t
WHERE  length(btrim(t.b, '0…9')) < length(t.b)   p1
   OR  t.a % 1000 <> 0                            p2
```



  - With Boolean shortcut it makes a difference which disjunct is evaluated first. (Both predicates not selective, p_1: 85.9%, p_2: 99.9% of 10^7 rows pass.)

⇒ Many optimizer decisions indeed *are* **cost-based.**

2 ¦ Q_7 — Predicate (or Filter) Evaluation



```
SELECT t.a, t.b
FROM   ternary AS t
WHERE  t.a % 2 = 0 AND [OR] t.c < 1  -- either AND or OR
```

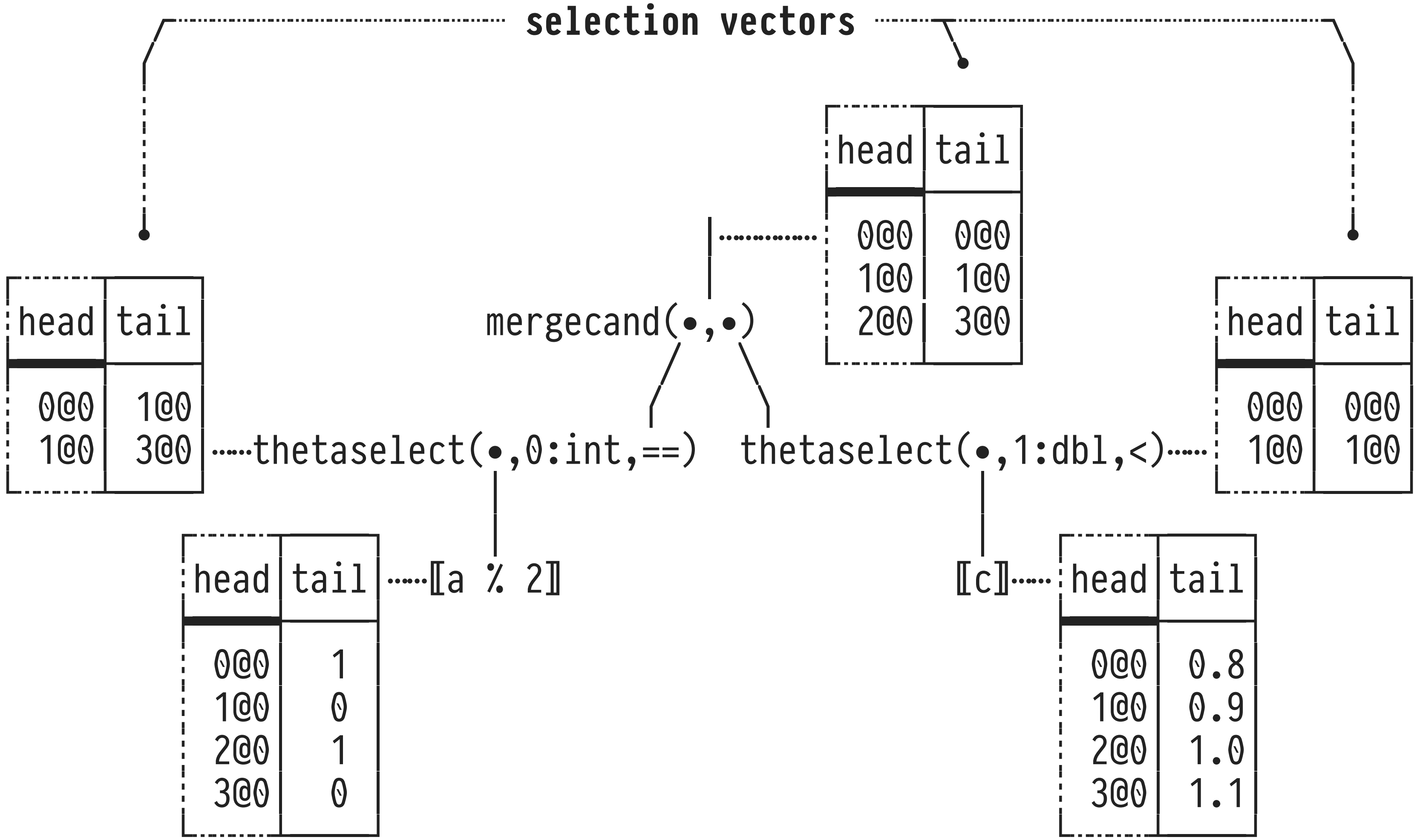
MonetDB can evaluate basic predicates on individual column BATs (here: a and c) ❶ but then needs to

1. derive the result of composite predicates ❷ and
2. propagate the filter effect to all output columns (here: a, b) ❸ to form the final selection result.

Using EXPLAIN on Q_7 (Boolean Connective: OR)

```
sql> EXPLAIN SELECT t.a, t.b
             FROM   ternary AS t
             WHERE  t.a % 2 = 0 OR t.c < 1;
   ⋮
  ternary :bat[:oid] := sql.tid(sql, "sys", "ternary");
  a0      :bat[:int] := sql.bind(sql, "sys", "ternary", "a", 0:int);
  a       :bat[:int] := algebra.projection(ternary, a0);
  e1      :bat[:int] := batcalc.%(a, 2:int);                ← a % 2
❶ p1      :bat[:oid] := algebra.thetaselect(e1, 0:int, "=="); ← p1 ≡ a % 2 = 0
  c0      :bat[:dbl] := sql.bind(sql, "sys", "ternary", "c", 0:int);
  c       :bat[:dbl] := algebra.projection(ternary, c0);
❶ p2      :bat[:oid] := algebra.thetaselect(c, 1:dbl, "<");    ← p2 ≡ c < 1
❷ or      :bat[:oid] := bat.mergecand(p1, p2);                 ← p1 ∨ p2
  b0      :bat[:str] := sql.bind(sql, "sys", "ternary", "b", 0:int);
❸ bres    :bat[:str] := algebra.projectionpath(or, ternary, b0); ← result col b
❸ ares    :bat[:int] := algebra.projection(or, a);             ← result col a
   ⋮
```

Result of a Predicate ≡ Selection Vectors



Selection Vectors (also: Candidate Lists)

- **Selection vector** sv: BAT of type `bat[:oid]`.
  i`@0` $\in sv \Leftrightarrow i$th input row satisfies filter predicate.

- Use `algebra.projection(`sv`, `col`)` to propagate filter effect to column col.

- Implement Boolean connectives for predicate p_i with sv_i:
  - p_1 OR p_2: `algebra.projection(bat.mergecand(`sv_1`,`sv_2`),•)`
  - p_1 AND p_2: `algebra.projectionpath(`sv_2`,`sv_1`,•)` with

    `algebra.projectionpath(`sv_2`,`sv_1`,•)` ≡
    `algebra.projection(`sv_2`, algebra.projection(`sv_1`,•))`.

Under a layer of C macros, the core of MonetDB's filtering routine *sv* := thetaselect(*col*:bat[:int],*v*:int,*θ*) resembles:

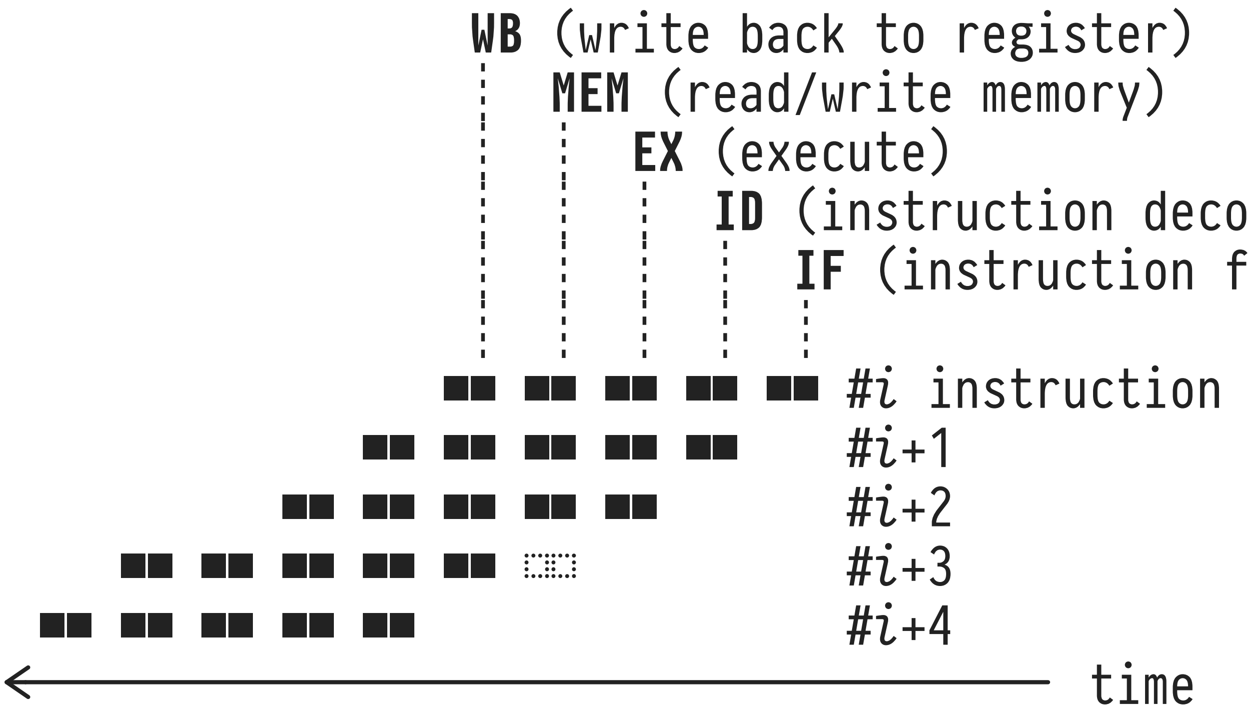
```
int thetaselect(int *sv, int *col, int v, θ)
{
  int SIZE = <number of rows in col>;                       /* input cardinality */
  int out = 0;

  for (int i = 0; i < SIZE; i += 1) {
      if (col[i] θ v) {                                  /* test filter condition */
          sv[out] = i;                                 /* build selection vector */
          out += 1;
      }
  }

  return out;                                             /* output cardinality */
}
```

Instruction Pipelining in Modern CPUs

Control flow branches (for, but particularly if) are a challenge for modern pipelining CPUs:

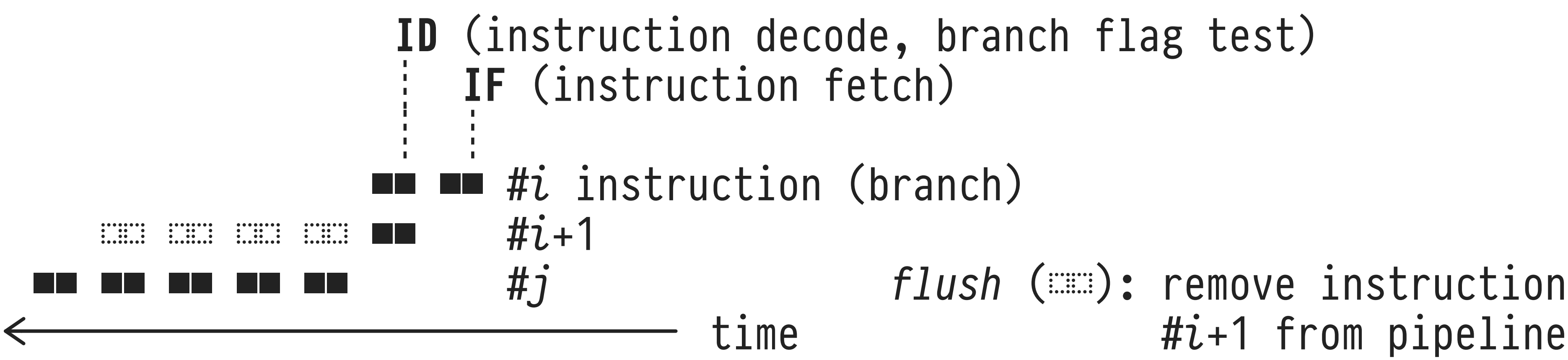


stall ( ): ins #*i*+3 cannot fetch its opcode (**IF**) while ins #*i* accesses memory (**MEM**)

Branch Taken? Yes, Flush Pipeline

This pipeline decides the outcome of branch *#i* (end of **ID**) only *after* instruction *#i*+1 has already been fetched (**IF**):

- If the branch is taken, **flush** instruction *#i*+1 from pipeline 👎, instead fetch instruction *#j* at jump target:



Branch Prediction: History and Heuristics

CPUs thus try to **predict the outcome of a branch #*i*** based on **earlier recorded outcomes** of the same branch:

Branch prediction	Fetch instruction
taken	#*j*
not taken	#*i*+1

- Also: **heuristics** based on typical control flow patterns:

Predicted *taken*

```
loop: …
  ⋮
  ⋮
  jcc loop
```

Predicted *not taken*

```
  jcc break
  ⋮
  jmp loop
break: …
```

Avoiding Branch Mispredictions

- A **mispredicted branch** 👎 leads to

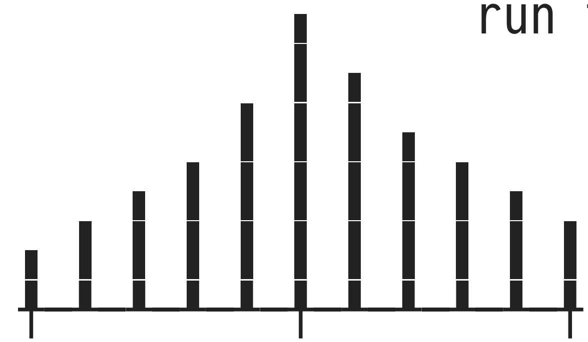
  1. pipeline flushes—effectively a stall—and
  2. (possibly) CPU instruction cache misses.

- The resulting runtime penalty indeed is significant ⇒ DBMSs aim to avoid branch mispredictions in tight inner loops:

  - prefer branch-less implementations of query logic,
  - reduce number of random/hard-to-predict branches.

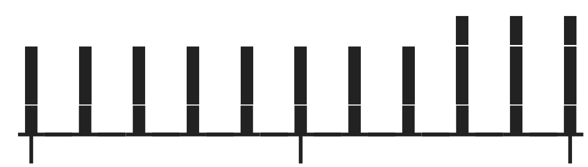
MonetDB: Branch-Less Selection ❷

❶
```
for (int i = 0; i < SIZE; i += 1) {
    if (col[i] < v) {
        sv[out] = i;
        out += 1;
    }
}
```



❷
```
for (int i = 0; i < SIZE; i += 1) {
    sv[out] = i;
    out += (col[i] < v);
}
```

(col[i] < v) ≡ 1 if predicate satisfied, else 0



❷: Only well-predictable loop control flow (for) remains.

Mixed-Mode Selection

There is an entire space of possibilities to implement composite predicates (e.g., the conjunction p_1 AND p_2):

- Use branch-less selection via `out += ` p_1 `&` p_2 (note use of C's bit-wise *and* operator `&`).
- Identify the *more selective*[1] (and thus more predictable) conjunct p_1, say, then use

```
if (p₁) {
    sv[out] = i;
    out += (p₂);
}
```

[1] **This is important.** Using `if (`p_2`) …` instead, where p_2 is unpredictable, immediately ruins the plan.