DB 2

08 – Predicate Evaluation

Summer 2018

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 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=5 …)
  Filter: ((c < '1'::double precision) AND ((a % 2) = 0))
  Rows Removed by Filter: 995
Planning time: 2.125 ms
Execution time: 1.894 ms
```

- Filter predicate evaluated during Seq Scan.
- Estimated **selectivity** of predicate $^1/_{1000}$ (real: $^5/_{1000}$).

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.**

PostgreSQL source code (expression interpreter):

- Boolean shortcut in BOOL_AND_STEP_FIRST: src/backend/executor/execExpInterp.c, function ExecInterpExpr()
- Invocation of expression interpreter in Seq Scan: src/backend/executor/execScan.c, function ExecScan(). See call ExecQual(qual, econtext), preceded by comment:

```
/*
 * check that the current tuple satisfies the qual-clause
 *
 * check for non-null qual here to avoid a function call to ExecQual()
 * when the qual is null ... saves only a few cycles, but they add up
 * ...
 */
if (qual == NULL || ExecQual(qual, econtext))
[...]
```

- Function ExecQual() found in src/include/executor/executor.h. Can process entire list *ps* of conjunctive clauses:
    - ExecQual() handles case ps = [] and immediately returns *true* (*ex falso quod libet*).

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., $\mathrm{NOT}(\mathrm{NOT}(p)) \equiv p$ and $(p \text{ AND } q) \text{ OR } (p \text{ AND } r) \equiv p \text{ AND } (q \text{ OR } r)$).
  - Remove duplicate clauses (e.g., $p \text{ AND } p \equiv p$)
  - Apply De Morgan's laws.

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

Show heuristic predicate simplifcation at work:

```
-- ❶ Remove double NOT() + De Morgan:

db2=# EXPLAIN VERBOSE
        SELECT t.a, t.b
        FROM   ternary AS t
        WHERE  NOT(NOT(NOT(t.a % 2 = 0 AND t.c < 1)));
```

QUERY PLAN

Seq Scan on public.ternary t (cost=0.00..27.50 rows=1000 width=37)
  Output: a, b
  Filter: (((t.a % 2) <> 0) OR (t.c >= '1'::double precision))

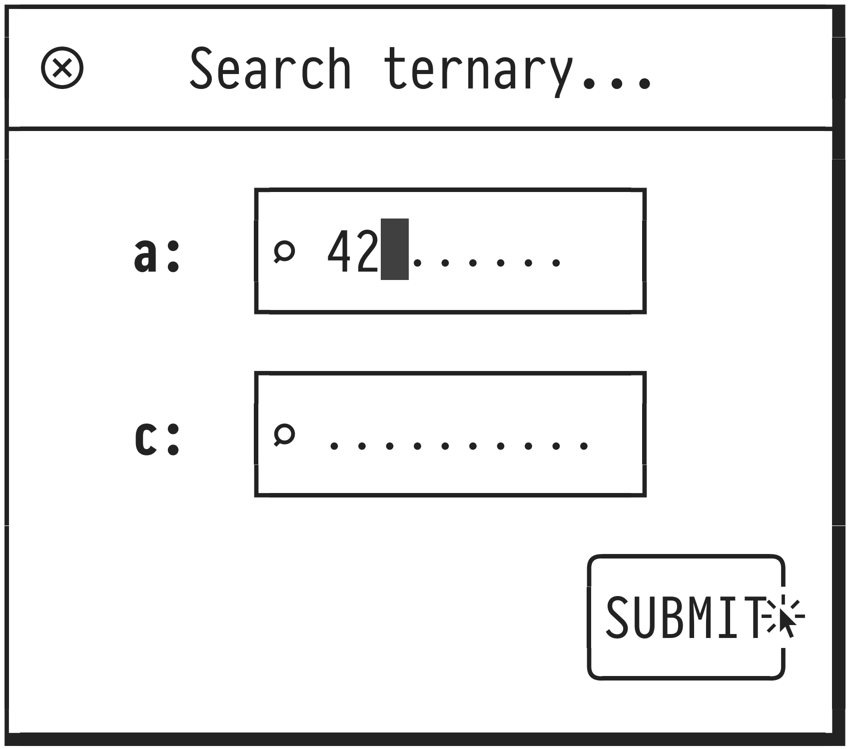
```
-- ❷ Inverse distributivity of AND:

db2=# EXPLAIN VERBOSE
        SELECT t.a, t.b
        FROM   ternary AS t
        WHERE (t.a % 2 = 0 AND t.c < 1) OR (t.a % 2 = 0 AND t.c > 2);
```

QUERY PLAN

Seq Scan on public.ternary t (cost=0.00..30.00 rows=5 width=37)
  Output: a, b
  Filter: (((t.c < '1'::double precision) OR (t.c > '2'::double precision)) AND ((t.a % 2) = 0))

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)
```

```
db2=# EXPLAIN VERBOSE
        SELECT t.*
        FROM   ternary AS t
        WHERE  (t.a = 42 OR 42 IS NULL)
          AND  (t.c = NULL OR NULL IS NULL);
```

QUERY PLAN

Seq Scan on public.ternary t (cost=0.00..22.50 rows=1 width=45)
  Output: a, b, c
  Filter: (t.a = 42)

```
db2=# EXPLAIN VERBOSE
        SELECT t.*
        FROM   ternary AS t
        WHERE  (t.a = NULL OR NULL IS NULL)
          AND  (t.c = NULL OR NULL IS NULL);
```

QUERY PLAN
Seq Scan on public.ternary t (cost=0.00..20.00 rows=1000 width=45)   Output: a, b, c

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**.

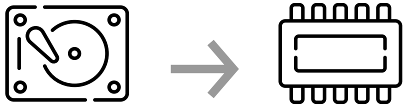
Selectivity-based decisions are important in query optimization, but selectivity is secondary for this example. Predicate evaluation cost dominates the overall query evaluation cost:

```
db2=# EXPLAIN ANALYZE
       SELECT *
       FROM ternary_10m AS t
       WHERE length(btrim(t.b, '01234567890')) < length(t.b)
          OR t.a % 1000 <> 0;
```

QUERY PLAN
Seq Scan on ternary_10m (cost=0.00..342594.10 rows=9966711 width=45) (actual time=0.043..14219.828 rows=9998573 loops=1)
Filter: ((length(btrim(b, '01234567890'::text)) < length(b)) OR ((a % 1000) <> 0))
Rows Removed by Filter: 1427
Planning time: 0.156 ms
Execution time: 14741.087 ms ◀

```
db2=# EXPLAIN ANALYZE
       SELECT *
       FROM   ternary_10m AS t
       WHERE  t.a % 1000 <> 0
          OR  length(btrim(t.b, '01234567890')) < length(t.b);
```

QUERY PLAN
Seq Scan on ternary_10m (cost=0.00..342594.10 rows=9966711 width=45) (actual time=0.025..2302.257 rows=9998573 loops=1)
Filter: (((a % 1000) <> 0) OR (length(btrim(b, '01234567890'::text)) < length(b)))
Rows Removed by Filter: 1427
Planning time: 0.116 ms
Execution time: 2803.925 ms ◀



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

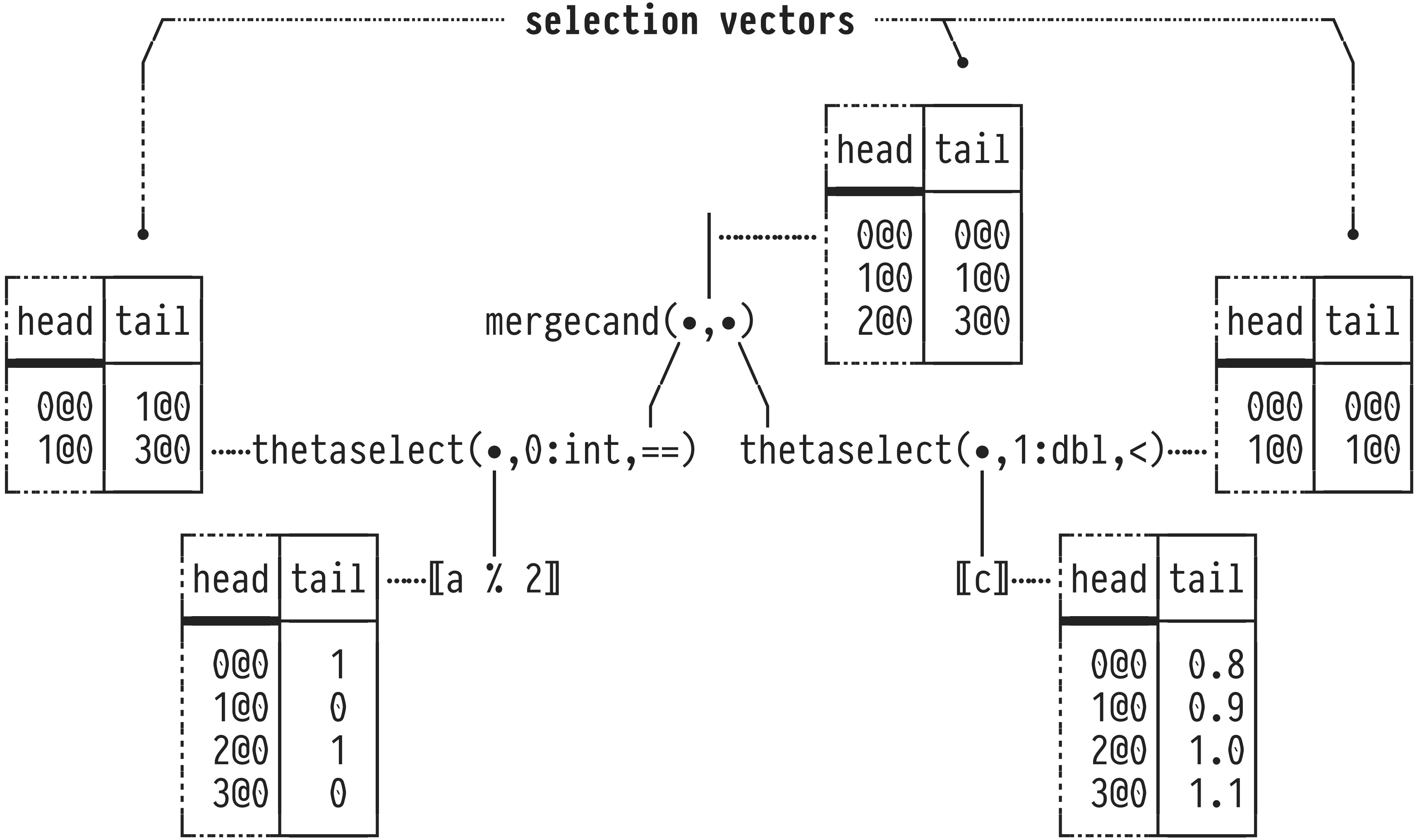
MonetDB can evaluate basic predicates on individual column BATs (here: a and c) ❶ but then needs to

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 ∈ sv ⇔ ith 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: `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,•`))`.

Demonstrate evaluation of disjunctive condition in MAL:

```
$ mclient -d scratch -l mal

  sql.init();
  sql := sql.mvc();
  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
  io.print(e1);
#--------------------------#
# h t  # name
# void  int  # type
#--------------------------#
[ 0@0,  1  ]
[ 1@0,  0  ]
[ 2@0,  1  ]
[ 3@0,  0  ]
[...]
  p1     :bat[:oid] := algebra.thetaselect(e1, 0:int, "==");  # ⬅ p₁ ≡ a % 2 = 0
  io.print(p1);
#--------------------------#
# h t  # name
# void  oid  # type
#--------------------------#
[ 0@0,  1@0  ]
[ 1@0,  3@0  ]
[...]
[ 498@0,  997@0  ]
[ 499@0,  999@0  ]
  c0     :bat[:dbl] := sql.bind(sql, "sys", "ternary", "c", 0:int);
  c      :bat[:dbl] := algebra.projection(ternary, c0);
  p2     :bat[:oid] := algebra.thetaselect(c, 1:dbl, "<");   # ⬅ p₂ ≡ c < 1
  io.print(p2);
#--------------------------#
# h t  # name
# void  void  # type
#--------------------------#
[ 0@0,  0@0  ]               ⬅ c = 0
[ 1@0,  1@0  ]               ⬅ c = 0.6931471805599453

# ----------------------------------------------------------------------
# Interlude on range selection
#
#                  v <!--⩽ hi (false ≡ <)    complement result?
```

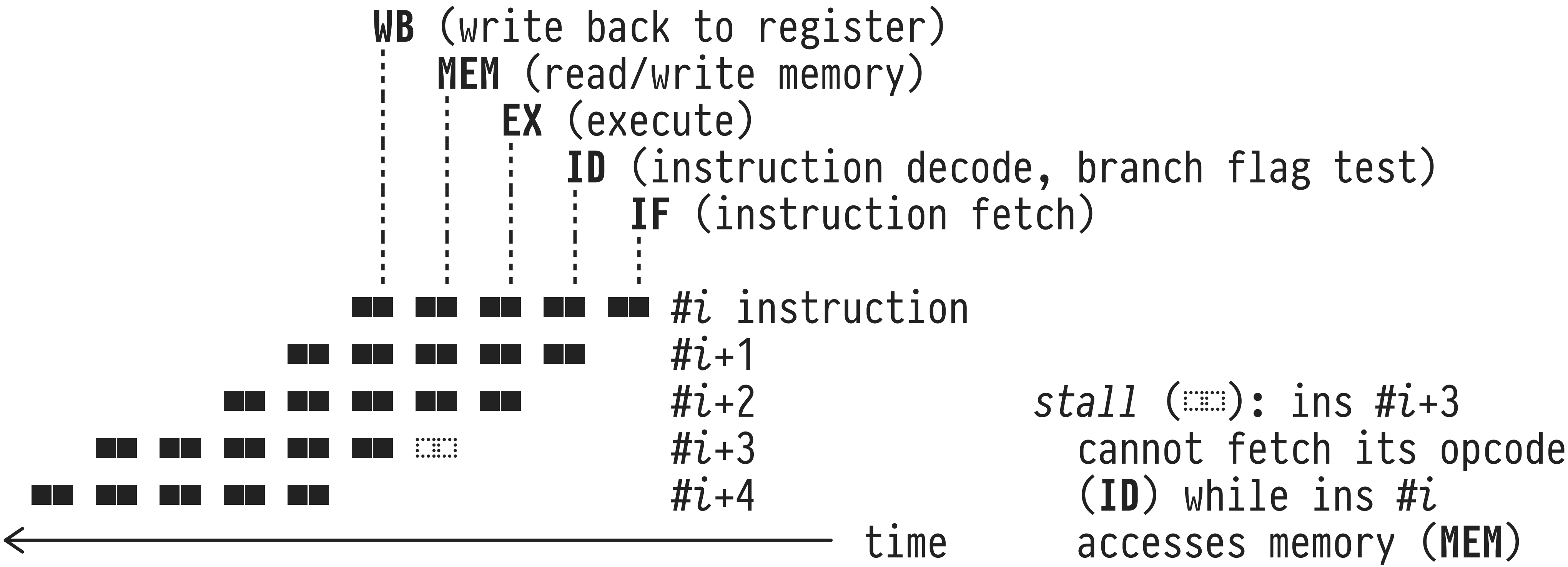
```
#
# algebra.select(col, lo, hi, true, false, false):
#
#              lo </⩽ v (true ≡ ⩽)
#
#
# Return oids of values v in col in the range [lo, hi)

  sv := algebra.select(c, 2:dbl, 2.5:dbl, true, false, false);  # ⬅ 2 ⩽ c < 2.5
  io.print(sv);
#--------------------------#
# h t  # name
# void  void  # type
#--------------------------#
[ 0@0,  7@0  ]
[ 1@0,  8@0  ]
[ 2@0,  9@0  ]
[ 3@0,  10@0  ]
[ 4@0,  11@0  ]
  range := algebra.projection(sv, c);
  io.print(range);
# ---------------------------------------------------------------------


  or := bat.mergecand(p1, p2);        #  ⬅ p₁ ∨ p₂
  io.print(or);
#--------------------------#
# h t  # name
# void  oid  # type
#--------------------------#
[ 0@0,  0@0  ]   ⬅ contributed by e3
[ 1@0,  1@0  ]   ⬅ contributed by e2 and e3
[ 2@0,  3@0  ]   ⬅ contributed by e2
[...]
  b0     :bat[:str] := sql.bind(sql, "sys", "ternary", "b", 0:int);
  bres   :bat[:str] := algebra.projectionpath(or, ternary, b0);  # ⬅ applies visibility in ternary AND THEN selection vector
  ares   :bat[:int] := algebra.projection(or, a);                # ⬅ applies selection vector
  io.print(ares, bres);
#--------------------------#
# t t t  # name
# void  int str  # type
#--------------------------#
[ 0@0,  1,  "c4ca4238a0b923820dcc509a6f75849b"  ]
[ 1@0,  2,  "c81e728d9d4c2f636f067f89cc14862c"  ]
[...]
[ 499@0,  998,  "9ab0d88431732957a618d4a469a0d4c3"  ]
[ 500@0,  1000, "a9b7ba70783b617e9998dc4dd82eb3c5"  ]
```

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

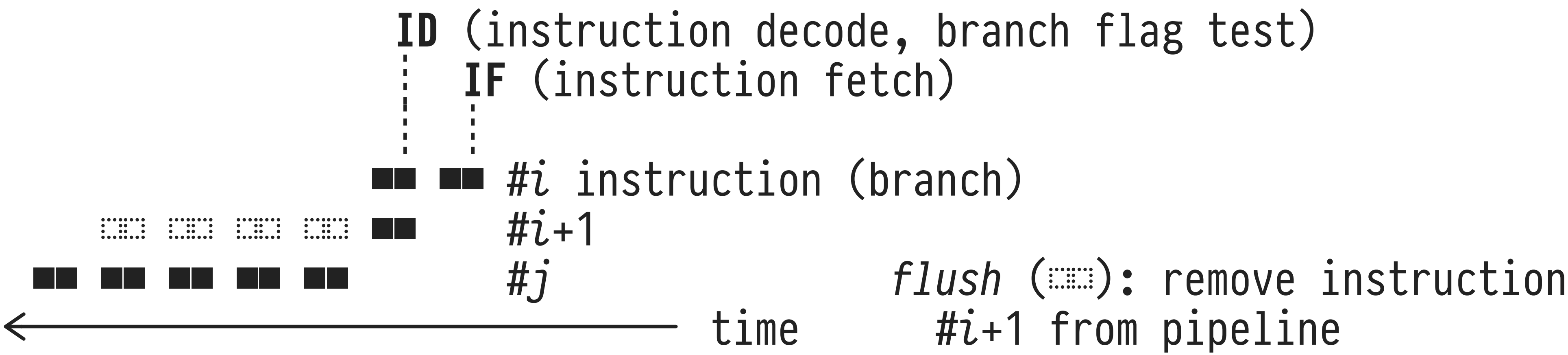
Control flow branches (`for`, but particularly `if`) are a challenge for modern pipelining CPUs:



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) 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.

Demonstrate the effects of branch misprediction using a selection col < v. Linearly grow v from 0...max. Selectivity of predicate col < v linearly grows from 0% to 100%.

See file mat/branch-prediction.c, core tight loop:

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

Compile via (⚠️ no -O or -O2 here):

```
$ cc -Wall  branch-prediction.c -o branch-prediction
```

QUIZ: Let students speculate about outcome of experiment.

❶ Sample run:

```
$ ./branch-prediction
 0 (selectivity:   0.00%) 33207µs   ⬅ predicate always false: branch always taken (perfect to predict, also meets heuristic)
 1 (selectivity:  10.00%) 55499µs   ⎫ branch outcome increasingly random and thus hard to predict,
 2 (selectivity:  20.00%) 82161µs   ⎬ branch mispredictions increasingly likely
 3 (selectivity:  29.99%) 107224µs  ⎭
 4 (selectivity:  40.00%) 153250µs  ⎫
 5 (selectivity:  49.99%) 128811µs  ⎬ branch outcome essentially arbitrary and thus impossible to predict
 6 (selectivity:  59.98%) 120362µs  ⎭
 7 (selectivity:  69.99%) 103390µs  ⎫ branch becomes more and more like to be untaken,
 8 (selectivity:  80.01%) 83365µs   ⎬ branch mispredictions decrease
 9 (selectivity:  90.01%) 65245µs   ⎭
10 (selectivity: 100.00%) 48755µs   ⬅ predicate always true: branch never taken (perfect to predict)
```

❷ Experiment (1): **Sort** the column vector in ascending order. Column vector is effectively divided into two parts:

1. starting region of column vector where always col < v: branch never taken ⇒ perfect to predict
2. ending region of column vector where always col > v: branch always taken ⇒ perfect to predict

(Actual value of v immaterial—the larger v, the longer sv grows which needs a bit of work.)

Sample run for Experiment (1):

```
 0 (selectivity:   0.00%) 27578µs   ⬅ builds empty selection vector, no memory writes
 1 (selectivity:  10.00%) 35127µs
 2 (selectivity:  20.00%) 37500µs
 3 (selectivity:  29.99%) 40296µs
 4 (selectivity:  40.00%) 39552µs
 5 (selectivity:  49.99%) 41864µs
 6 (selectivity:  59.98%) 43514µs
 7 (selectivity:  69.99%) 43268µs
 8 (selectivity:  80.01%) 43903µs
 9 (selectivity:  90.01%) 46592µs
10 (selectivity: 100.00%) 50677µs   ⬅ builds long selection vector, lots of memory writes
```

3 Experiment (2): A **branch-less** selection.

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

Sample run for Experiment (2):

```
 0 (selectivity:   0.00%) 45288µs ⎫  ⬅ repeatedly writes to sv[0] only
 1 (selectivity:  10.00%) 46625µs ⎪
 2 (selectivity:  20.00%) 49592µs ⎪
 3 (selectivity:  29.99%) 47187µs ⎪
 4 (selectivity:  40.00%) 49891µs ⎬  run time unaffacted by predicate selectivity
 5 (selectivity:  49.99%) 49738µs ⎪
 6 (selectivity:  59.98%) 49811µs ⎪
 7 (selectivity:  69.99%) 46630µs ⎪
 8 (selectivity:  80.01%) 47391µs ⎪
 9 (selectivity:  90.01%) 47367µs ⎪
10 (selectivity: 100.00%) 52009µs ⎭  ⬅ writes to entire sv[0...SIZE-1]
```

File mat/branch-prediction.c:

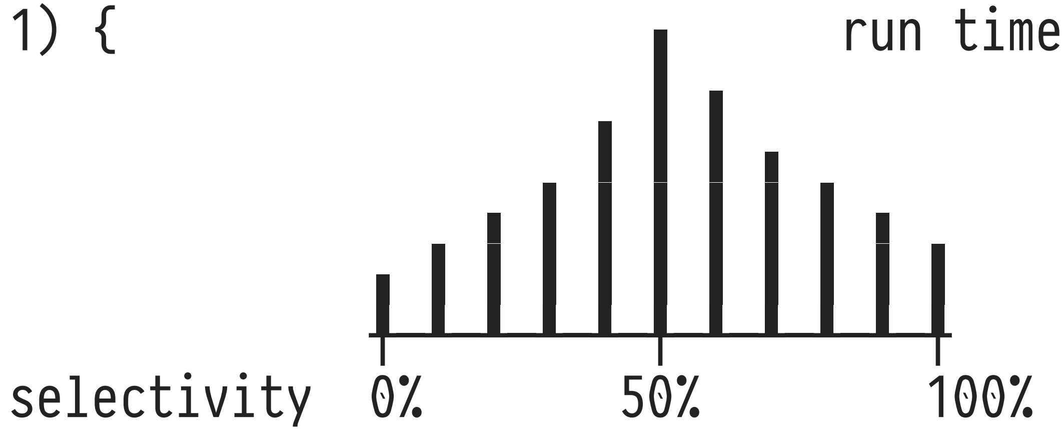
```
/* Demonstrate the effects of branch mispredictions for a selection
 * col < v implemented in a tight loop
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys time.h="">
#include <assert.h>

#define MICROSECS(t) (1000000 * (t).tv_sec + (t).tv_usec)
```

```
#define SIZE (16 * 1024 * 1024)
#define STEPS 11

/* comparison of a, b (used in qsort) */
int cmp (const void *a, const void *b)
{
    return *((int*) a) - *((int*) b);
}

int main()
{
    int *col;        /* column vector */
    int *sv;         /* selection vector */

    int out;
    float selectivity;

    struct timeval t0, t1;
    unsigned long duration;

    /* allocate memory */
    col = malloc(SIZE * sizeof(int));
    assert(col);
    sv = malloc(SIZE * sizeof(int));
    assert(sv);

    /* initialize column with (pseudo) random values in interval 0...RAND_MAX */
    srand(42);
    for (int i = 0; i < SIZE; i += 1) {
        col[i] = rand();
    }

    /* Experiment (1):
     */
    // qsort(col, SIZE, sizeof(int), cmp);

    for (int step = 0; step < STEPS; step += 1) {

        /* v grows linearly 0...RAND_MAX in STEPS steps */
        int v = step * (RAND_MAX / (STEPS - 1));

        gettimeofday(&t0, NULL);

        out = 0;
        for (int i = 0; i < SIZE; i += 1) {
```

```
            if (col[i] < v) {
                sv[out] = i;
                out += 1;
            }

            /* Experiment (2):
             * a branch-less copy
             */
            // sv[out] = i;
            // out += (col[i] < v);
        }

        gettimeofday(&t1, NULL);
        duration = MICROSECS(t1) - MICROSECS(t0);

        selectivity = ((float)out / SIZE) * 100.0;

        printf ("%2u (selectivity: %6.2f%%)\t%luµs\n",
                step, selectivity, duration);
    }

    return 0;
}
```

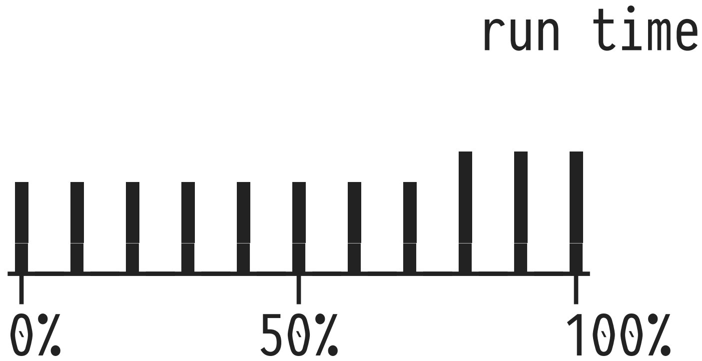
MonetDB: Branch-Less Selection ❷

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);
}
```

≡ 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.

Demonstrate three alternatives for the implementation of the composite predicate `col < v AND col % 2 = 0`.

See file `mat/mixed-mode-conjunction.c`. Compile via (⚠ no `-O2` here):

```
$ cc -Wall mixed-mode-conjunction.c -o mixed-mode-conjunction
```

- (A) Branch-less selection performs independent of selectivity of both predicates.
- (B) Mixed-mode selection with selective/predictable `col < v` in `if (col < v)` performs better than branch-less as long as `col < v` is very selective (saves work).
- (C) Mixed-mode selection with unpredictable `col % 2 = 0` in `if (col % 2 == 0)` always performs the worst.

Sample run:

```
$ ./mixed-mode-conjunction
    sel      A      mixed B  mixed C
 0   0.00%   87ms   28ms     208ms
 1   5.00%   87ms   58ms     201ms
 2  10.00%   86ms   77ms     198ms
 3  15.00%   84ms   101ms    200ms
 4  20.00%   86ms   125ms    208ms
 5  25.00%   83ms   150ms    205ms      ← selectivity of (col < v) = 50%
 6  30.00%   85ms   135ms    195ms
 7  35.00%   82ms   122ms    200ms
 8  40.00%   82ms   108ms    198ms
 9  45.00%   81ms   108ms    209ms
10  50.00%   85ms   87ms     202ms
      ↑
 selectivity of col < v AND col % 2 = 0
```

File `mat/mixed-mode-conjunction.c`:

```
/* Demonstrate alternatives for the implementation of
 * conjunctive predicate col < v ∧ col % 2 = 0:
 *
 * (A) branch-less selection (via & and +=)
 * (B) mixed mode selection (via if [varying selectivity] and +=)
 * (C) mixed mode selection (via if [unpredictable] and +=)
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys time.h="">
#include <assert.h>

#define MILLISECS(t) ((1000000 * (t).tv_sec + (t).tv_usec) / 1000)
```

```
#define SIZE (16 * 1024 * 1024)
#define STEPS 11

/* comparison of a, b (used in qsort) */
int cmp (const void *a, const void *b)
{
    return *((int*) a) - *((int*) b);
}

int main()
{
    int *col;     /* column vector */
    int *sv;      /* selection vector */

    int out;
    float selectivity;

    struct timeval t0, t1;
    unsigned long duration1, duration2, duration3;

    /* allocate memory */
    col = malloc(SIZE * sizeof(int));
    assert(col);
    sv = malloc(SIZE * sizeof(int));
    assert(sv);

    /* initialize column with (pseudo) random values in interval 0...RAND_MAX */
    srand(42);
    for (int i = 0; i < SIZE; i += 1) {
        col[i] = rand();
    }

    /* Quiz: how will sorting the column affect run time?
    */
    // qsort(col, SIZE, sizeof (*col), cmp);

    printf("\tsel\tA\tmixed B\tmixed C\n");

    for (int step = 0; step < STEPS; step += 1) {

        /* v grows linearly 0...RAND_MAX in STEPS steps */
        int v = step * (RAND_MAX / (STEPS - 1));

        /* ----------  alternative A ---------- */

        gettimeofday(&t0, NULL);
```

```
        out = 0;
        for (int i = 0; i < SIZE; i += 1) {
            sv[out] = col[i];
            out += ((col[i] < v) & (col[i] % 2 == 0));
        }

        gettimeofday(&t1, NULL);
        duration1 = MILLISECS(t1) - MILLISECS(t0);


        /* ---------- alternative B ---------- */

        gettimeofday(&t0, NULL);

        out = 0;
        for (int i = 0; i < SIZE; i += 1) {
            if (col[i] < v) {
                sv[out] = col[i];
                out += (col[i] % 2 == 0);
            }
        }

        gettimeofday(&t1, NULL);
        duration2 = MILLISECS(t1) - MILLISECS(t0);


        /* ---------- alternative C ---------- */

        gettimeofday(&t0, NULL);

        out = 0;
        for (int i = 0; i < SIZE; i += 1) {
            if (col[i] % 2 == 0) {
                sv[out] = col[i];
                out += (col[i] < v);
            }
        }

        gettimeofday(&t1, NULL);
        duration3 = MILLISECS(t1) - MILLISECS(t0);

        selectivity = ((float)out / SIZE) * 100.0;

        printf ("%2u\t%5.2f%%\t%lums\t%lums\t%lums\n",
                step, selectivity, duration1, duration2, duration3);
    }
```

```
    return 0;
}
```

⚠ File mat/mixed-mode-selection.c contains even more implementation alternatives, turn this into an assignment?