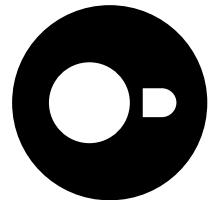


Dissecting the Duck's Innards



⑦

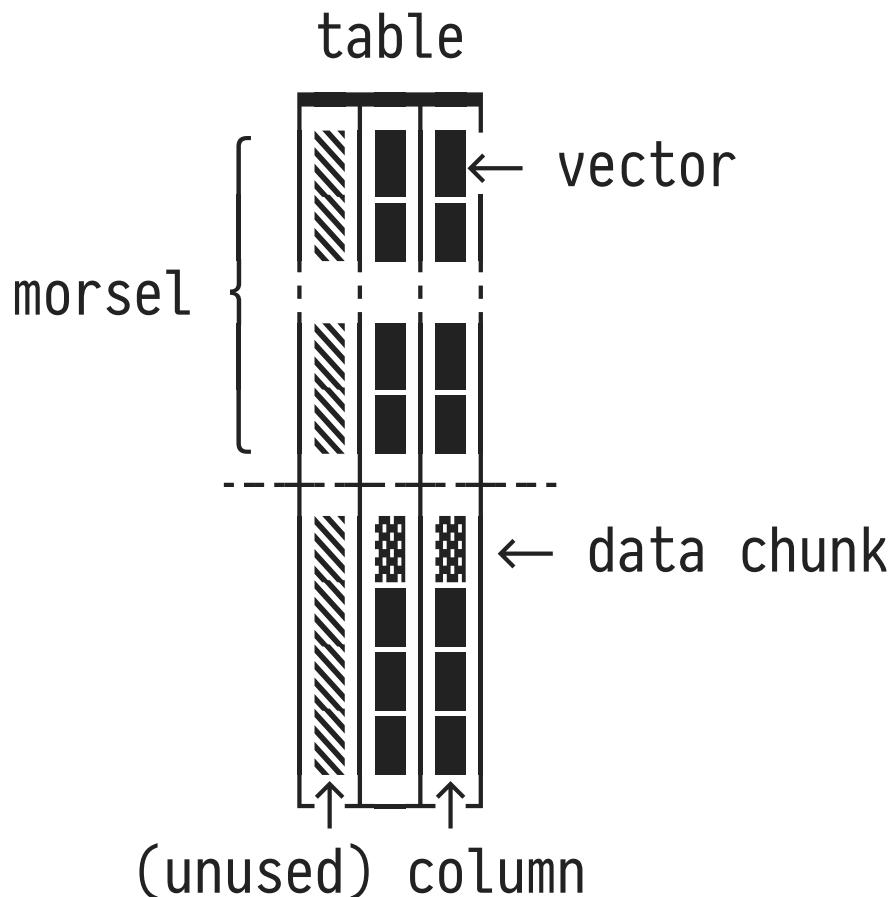
Vectorized Query Execution

Winter 2025/26

Torsten Grust
Universität Tübingen, Germany

1 | DuckDB Terminology Recap

Before we proceed, let us make sure that we agree on the data items processed by DuckDB's query engine:



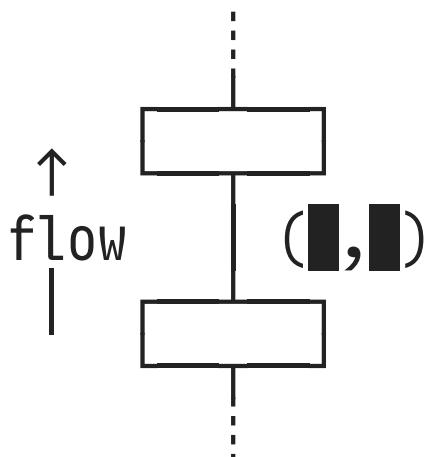
- **Tables** are organized in **columns**, some of which may be unused in any given plan operator.
- Columns are split into **vectors** , each containing 2048 values.¹
- A **data chunk** () groups vectors of multiple columns, representing a horizontal slice of a table.
- Threads consume input data **morsel**-by-**morsel**, each consisting of 60 data chunks (= 122,880 rows).

¹ DuckDB's vector size can be configured at compile time ([STANDARD_VECTOR_SIZE](#)).

2 | Passing Data Chunks Between Operators

During plan execution, operators process **one data chunk at a time**, then pass that chunk downstream:

Memory Location	Latency ✅	Size (Mac M2 Max)	... Can Hold
CPU Registers		$n \times 64$ bits	cell value(s)
L1 Cache	< 1 ns	(per ⚡) 192 KB	vector
L2 Cache	2.8 ns	(shared) 32 MB	data chunk
RAM 📁	≈ 100 ns	16–96 GB	column(s)



- The passed **data chunks** will fit in the L2 cache. Operators can access intermediate results with low latency. #023 👍
- Passing **entire columns**: many intermediates will only fit into RAM. High latency. 👎
- Passing **single rows**: frequent context switches between operators. CPU overhead. 👎

3 | DuckDB Vectors: Logical vs. Physical

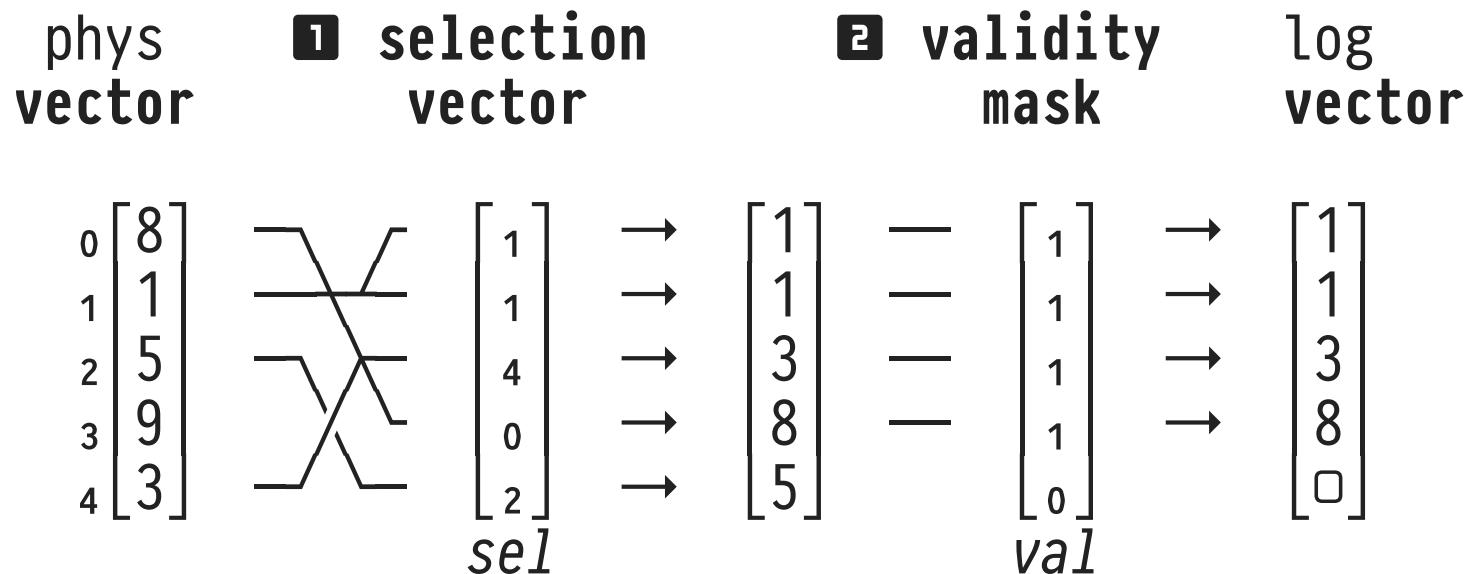
To form a horizontal table slice, data chunks ($\boxed{}, \dots, \boxed{}$) contain a **vector** for each relevant column.

- Vector $\boxed{}$ (**logical**): sequence of 2048 values of a *single type*.
- DuckDB exploits regularity in value sequences to implement a family of compressed **physical vector representations**:

FLAT	CONSTANT	DICTIONARY	SEQUENCE
$\begin{bmatrix} 1 \\ 9 \\ 0 \\ 4 \\ 2 \end{bmatrix}$ $\begin{bmatrix} 1 \\ 9 \\ 0 \\ 4 \\ 2 \end{bmatrix}$	$[42]$ $\begin{bmatrix} 42 \\ 42 \\ 42 \\ 42 \\ 42 \end{bmatrix}$ <i>const</i>	$^0\begin{bmatrix} \text{DE} \\ \text{NL} \end{bmatrix}$ $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ <i>dict</i>	$[0]$ $\begin{bmatrix} 0 \\ 10 \\ 20 \\ 30 \\ 40 \end{bmatrix}$ <i>base</i>
phys log	phys log	phys log	phys log

DuckDB Vectors: Selection Vectors and Validity Masks

Any vector is associated with a **selection vector** and **validity mask** used to permute/filter its value sequence and encode **NULLs** (□):

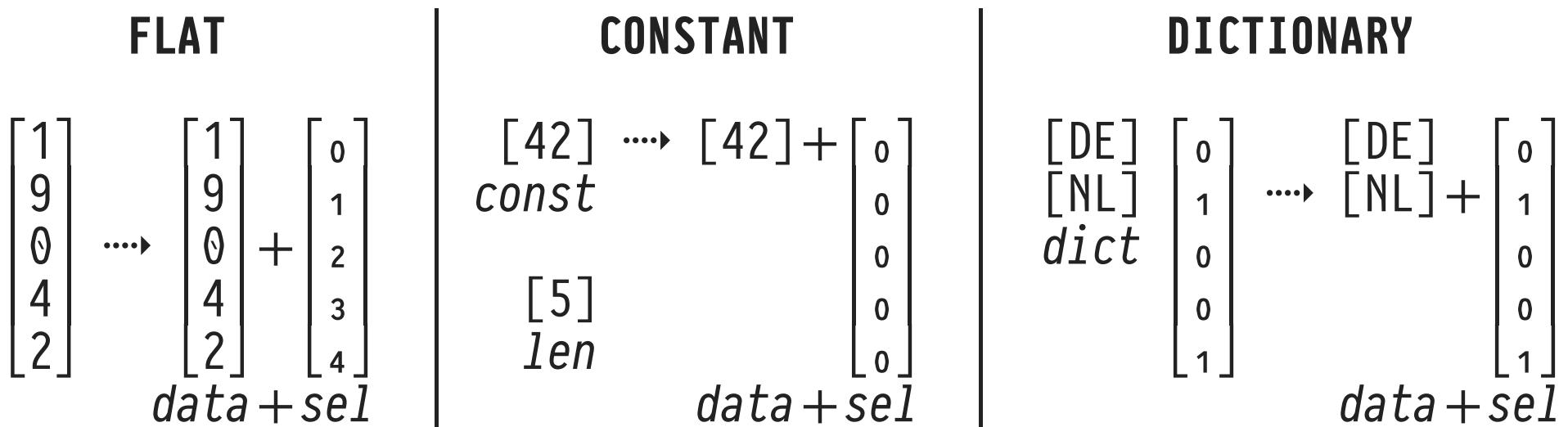


- Selection vectors contain vector indices $\in \{0, \dots, 2047\}$. Can help to avoid copying—potentially sizable—values after a sorting or filtering operation.
- Validity masks contain Booleans (0/1).

DuckDB Vectors: Unified Representation

DuckDB aims to push vectors in compressed physical representation from operator to operator. However, *handling all representations everywhere* would lead to combinatorial code explosion. 🤦

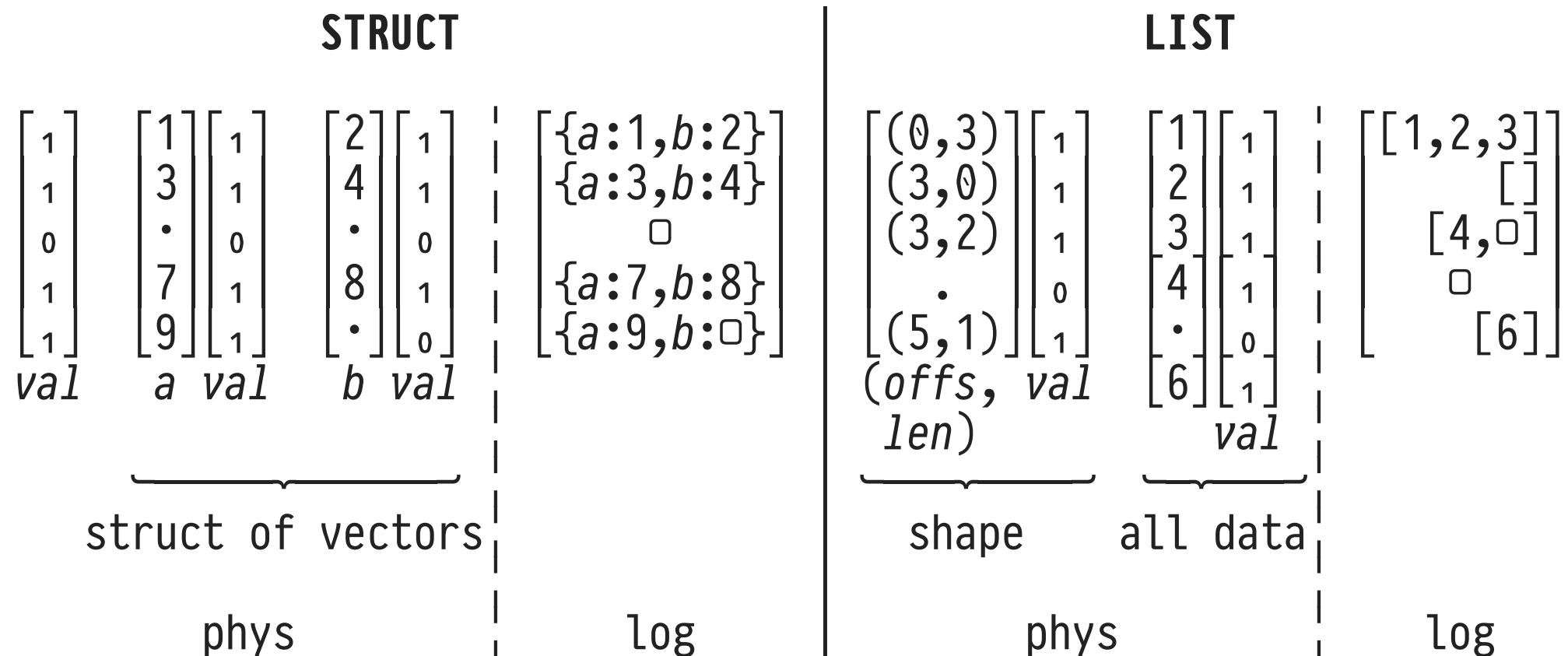
- 💡 Convert to (\rightarrow) and then process a **unified representation** comprising a *data vector+selection vector* pair.
- NB.** This conversion does *not* involve value copying:²



² SEQUENCE vectors are first expanded into FLAT vectors, then converted into the unified representation.

DuckDB Vectors of Complex Values (Structs {...}, Lists [...])

- **Vectors of structs** are represented as structs of vectors.
- **Vectors of lists** store list shapes and data values separately, combine data of all lists into a single data vector.



- Nested types: apply representation schemes recursively.

#025

4 | Super-Specific Vector Code

Q: How does DuckDB perform a binary operation on vectors `l` and `r`?

That depends on

- 1 the kind of operation (e.g., arithmetics: `+`, `*`, ...),
- 2 value types (`int`, `float`, ...),
- vector representations of 3 `l` and 4 `r` (`FLAT`, `CONST`, ...).

Q Branch on 1...4 before we enter the tight loop that scans the values in `l` and `r`:

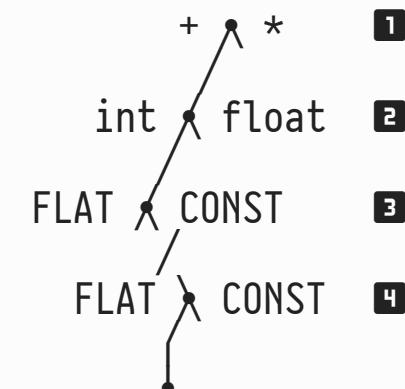
- Avoids repeated tests (with identical outcome),
- leads to *tight inner loops* that feature less branches.

Super-Specific Vector Code

```

switch (1 op)
  case +: switch (2 type)
    case int: switch (3 l.repr)
      case FLAT:   switch (4 r.repr)
        case FLAT:   res.repr = FLAT;
                      for (i = 0; i < count; i++)
                        res.data[i] = l.data[i] + r.data[i];
                      break;
        case CONSTANT: res.repr = FLAT;
                        for (i = 0; i < count; i++)
                          res.data[i] = l.data[i] + r.const;
                        break;
        ...
      case CONSTANT: switch (4 r.repr)
        case FLAT:   res.repr = FLAT;
                      for (i = 0; i < count; i++)
                        res.data[i] = l.const + r.data[i];
                      break;
        case CONSTANT: res.repr = CONSTANT;
                        res.const = l.const + r.const;
                        break;
        ...
    case float: ...
    ...
  case *: ...
  ...

```



} Super-specific code section:
Add the **constant** vector l of
integers to **flat** vector r

Super-Specific Vector Code

Previous slide does not handle **selection vector/validity mask** (each leaf of the decision tree \wedge requires code much like this):

```
for (i = 0; i < count; i++)      /* code for leaf 1: +, 2: int, 3: FLAT, 4: FLAT */
| lindex = l.sel[i];
| rindex = r.sel[i];
| if (l.val[lindex] && r.val[rindex])
| | res.data[i] = l.data[lindex] + r.data[rindex];
| | res.val[i]  = 1;
| else
| | res.val[i]  = 0;
```

DuckDB aims to control the resulting C++ code volume:

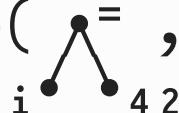
1. Convert vectors `l`, `r` to **unified representation**, process these by generic loops (cuts lower levels of \wedge).
2. Use **C++ templates** to generate code at *DBMS compile time*.³

³ A different breed of DBMS—for example [Umbra](#) —generates and compiles C++/IR/machine code at *query translation time*. These query-specific code fragments are then linked with the DBMS and invoked at runtime.

Deep Dive Into DuckDB C++ Code: Vector Operations ①

```
SELECT i = 42          -- i: SEQUENCE (base=1, inc=1), 42: CONST
FROM generate_series(1, 100) AS _(i)
```

Trace how DuckDB vectorizes the evaluation of expression `i = 42`:

ExpressionExecutor::Execute(, ..., count, result)

```
void ExpressionExecutor::Execute(
    const BoundComparisonExpression &expr, ..., idx_t count, Vector &result) {
    ...
    Execute(*expr.left, ..., count, left);   ↪ left ≡ [1|1]^s (SEQUENCE)
    Execute(*expr.right, ..., count, right);  ↪ right ≡ [42]^c (CONSTANT)

    switch (expr.GetExpressionType() ①) {
        case ExpressionType::COMPARE_EQUAL:
            VectorOperations::Equals(left, right, result, count);
            break;
        case ExpressionType::COMPARE_NOTEQUAL:
            VectorOperations::NotEquals(left, right, result, count);
            break;
        ...
    }
}
```

Deep Dive Into DuckDB C++ Code: Vector Operations ②

VectorOperations::Equals([1|1]^s, [42]^c, result, count)

```
void VectorOperations::Equals(
    Vector &left, Vector &right, Vector &result, idx_t count) {
    ComparisonExecutor::Execute<duckdb::Equals>(left, right, result, count); ④
}
```

```
struct ComparisonExecutor {
private:
    template <class T, class OP>
    static inline void TemplatizedExecute(
        Vector &left, Vector &right, Vector &result, idx_t count) {
        BinaryExecutor::ExecuteSwitch<T, T, bool, OP>(left, right, result, count); ⑤
    }
public:
    template <class OP>
    static inline void Execute(
        Vector &left, Vector &right, Vector &result, idx_t count) {
        :
        switch (left.GetType().InternalType() ⑥) {
            :
            case PhysicalType::INT64:
                TemplatizedExecute<int64_t, OP>(left, right, result, count); ⑦
                break;
            :
        }
    }
}
```

DuckDB 1.4 source: src/common/vector_operations/comparison_operators.cpp

Deep Dive Into DuckDB C++ Code: Vector Operations ③

`BinaryExecutor::ExecuteSwitch<int64_t, int64_t, bool, ..., duckdb::Equals, ...> ([1|1]s, [42]c, result, count, ...)`

```
struct BinaryExecutor {
    :
    template <class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, ..., class OP, ...>
    static void ExecuteSwitch(
        Vector &left, Vector &right, Vector &result, idx_t count, ...) {
        auto left_vector_type = left.GetVectorType();                      ↪ SEQUENCE_VECTOR
        auto right_vector_type = right.GetVectorType();                     ↪ CONSTANT_VECTOR
        if (left_vector_type == VectorType::CONSTANT_VECTOR ❸ &&
            right_vector_type == VectorType::CONSTANT_VECTOR ❹) {
            ExecuteConstant<LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE, ..., OP, ...>(
                left, right, result, ...);
        }
        :
        else {
            ExecuteGeneric<LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE, ..., OP, ...>(
                left, right, result, count, ...);
        }
    }
    :
}
```

DuckDB 1.4 source: `src/include/duckdb/common/vector_operations/binary_executor.hpp`

Deep Dive Into DuckDB C++ Code: Vector Operations ④

No specific code section for vector operation **SEQUENCE = CONSTANT**: first transform **left** and **right** into **unified representation**.

```
BinaryExecutor::ExecuteGeneric<int64_t, int64_t, bool, ..., duckdb::Equals, ...> (
    [1|1]^s, [42]^c, result, count, ...)
```

```
struct BinaryExecutor {
    ...
    template <class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, ..., class OP, ...>
    static void ExecuteGeneric(
        Vector &left, Vector &right, Vector &result, idx_t count, ...) {
        UnifiedVectorFormat ldata, rdata;

        left.ToUnifiedFormat(count, ldata);    ↪ ldata ≡ [[1,2,3,...,100]]^u (UnifiedVector)
        right.ToUnifiedFormat(count, rdata);   ↪ rdata ≡ [[42,42,...,42]]^u (UnifiedVector)

        result.SetVectorType(VectorType::FLAT_VECTOR);
        auto result_data = FlatVector::GetData<RESULT_TYPE>(result);
        ExecuteGenericLoop<LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE, ..., OP, ...> (
            UnifiedVectorFormat::GetData<LEFT_TYPE>(ldata),
            UnifiedVectorFormat::GetData<RIGHT_TYPE>(rdata),
            result_data, ..., count, ...);
    }
    ...
}
```

Deep Dive Into DuckDB C++ Code: Vector Operations ⑤

Both argument vectors [...]^u are now in unified representation:

```
BinaryExecutor::ExecuteGenericLoop<int64_t, int64_t, bool, ..., duckdb::Equals, ...> ( [1,2,3,...,100]u, [42,42,...,42]u, result, ..., count, ...)
```

```
struct BinaryExecutor {  
    :  
    template <class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, ..., class OP, ...>  
    static void ExecuteGenericLoop(  
        const LEFT_TYPE * __restrict ldata, const RIGHT_TYPE * __restrict rdata,  
        RESULT_TYPE * __restrict result_data, ..., idx_t count, ...)  
    {  
        :  
        for (idx_t i = 0; i < count; i++) {  
            auto lentry = ldata[i];  
            auto rentry = rdata[i];  
            result_data[i] = Operation<..., OP, LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE> (  
                ..., lentry, rentry, ...);  
        }  
    }  
    :  
}
```

DuckDB 1.4 source: [src/include/duckdb/common/vector_operations/binary_executor.hpp](https://github.com/duckdb/duckdb/blob/v1.4.0/src/include/duckdb/common/vector_operations/binary_executor.hpp)

Deep Dive Into DuckDB C++ Code: Vector Operations ⑥

BinaryExecutor::ExecuteGenericLoop<int64_t, int64_t, bool, ..., duckdb::Equals, ...> (
 [1,2,3,...,100]º, [42,42,...,42]º, result, ..., count, ...)

```
struct BinaryExecutor {
:
template <class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, ..., class OP, ...>
static void ExecuteGenericLoop(
    const LEFT_TYPE * __restrict ldata, const RIGHT_TYPE * __restrict rdata,
    RESULT_TYPE * __restrict result_data, ..., idx_t count, ...) {
    :
    for (idx_t i = 0; i < count; i++) {
        auto lentry = ldata[i];
        auto rentry = rdata[i];
        result_data[i] = lentry == rentry;
    }
}
:
}
```

DuckDB 1.4 source: src/include/duckdb/common/vector_operations/binary_executor.hpp

- **ldata/rdata:** plain C++ arrays of **LEFT_TYPE/RIGHT_TYPE** elements.
- **const ... * __restrict:** Hey C++, we never write to these arrays and they do not overlap in memory. Go on, optimize!

5 | Compiling Tight Loops

Tight loops over vectors form the core of DuckDB's query engine.

- Routine to subtract two flat integer vectors (⚠ simplified):

```
#define STANDARD_VECTOR_SIZE 2048  
01 #023  
  
void PROJECT_sub_int_col_int_col(  
    int* col1, int* col2, int *res, int* sel)  
{  
    int i;  
  
    if (sel) {  
        // skip discussion of selection vector for now  
    }  
  
    for (i = 0; i < STANDARD_VECTOR_SIZE; i += 1) {  
        res[i] = col1[i] - col2[i];  
    }  
}
```

Assembly Code for Tight Loops

Use `clang` (options `-O2 -fno-vectorize -fno-unroll-loops`) to generate code for x86-64 CPUs.⁴

- Register assignment:

`col1: %rdi, col2: %rsi, res: %rdx, i: %rax`

```
PROJECT_sub_int_col_int_col:
    xorl %eax, %eax          # %rax ←32 0
loop:
    movl (%rdi,%rax,4), %ecx # %ecx ←32 mem[%rdi + %rax]
    subl (%rsi,%rax,4), %ecx # %ecx ←32 %ecx -32 mem[%rsi + %rax]
    movl %ecx, (%rdx,%rax,4) # mem[%rdx + %rax] ←32 %ecx
    incq %rax
    cmpq $2048, %rax         # 2048 loop iterations
    jne loop                  # exit if %rax = 2048
    retq
```

- **NB.** One loop exit test per vector element computed.

⁴ Use Matt Godbolt's  [Compiler Explorer](#) ↗ to inspect assembly code generated by contemporary compilers.

Explicit Loop Unrolling

- Manually perform **loop unrolling** to
 1. improve the ratio (*useful work*)/(*loop exit test*),
 2. expose independent work that may be executed in `//:`

```
void PROJECT_sub_int_col_int_col(int* col1, int* col2, int* res) #026
{
    int i;

    for (i = 0; i + 3 < STANDARD_VECTOR_SIZE; i += 4) {
        res[i] = col1[i] - col2[i];
        res[i+1] = col1[i+1] - col2[i+1];
        res[i+2] = col1[i+2] - col2[i+2];
        res[i+3] = col1[i+3] - col2[i+3];
    }
}
```



independent, execute in
any order or even in `//`

- **NB.** This code does not handle the case of `STANDARD_VECTOR_SIZE mod 4 ≠ 0`.

#027

Loop Unrolling by the Compiler

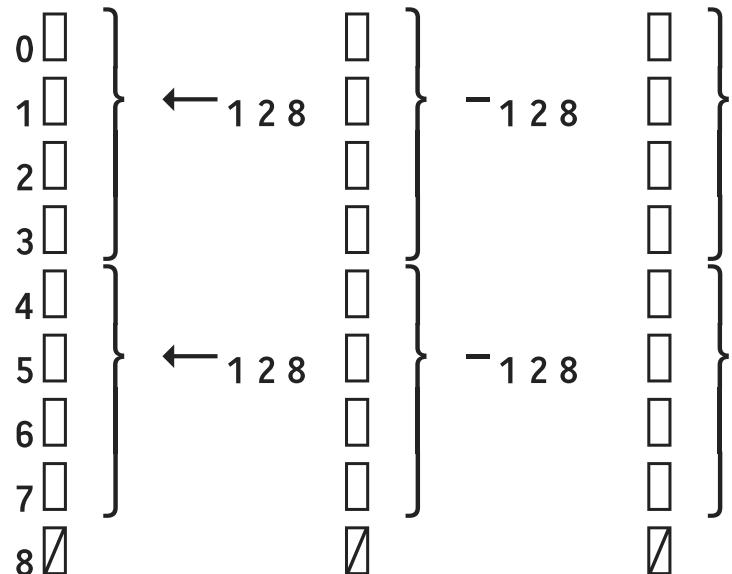
Compiler `clang` (options `-O2 -fno-vectorize -funroll-loops`) unrolls four loop iterations (easy for CPU to `//ize`):

```
PROJECT_sub_int_col_int_col:
    xorl %eax, %eax          # i ←32 0
loop:
    movl (%rdi,%rax,4), %ecx
    subl (%rsi,%rax,4), %ecx
    movl %ecx, (%rdx,%rax,4)
    movl 4(%rdi,%rax,4), %ecx
    subl 4(%rsi,%rax,4), %ecx
    movl %ecx, 4(%rdx,%rax,4)
    movl 8(%rdi,%rax,4), %ecx
    subl 8(%rsi,%rax,4), %ecx
    movl %ecx, 8(%rdx,%rax,4)
    movl 12(%rdi,%rax,4), %ecx
    subl 12(%rsi,%rax,4), %ecx
    movl %ecx, 12(%rdx,%rax,4)
    addq $4, %rax            # } i ←32 i+4
    cmpq $2048, %rax         # } 2048 / 4 = 512 loop iterations
    jne loop                 # exit if %rax = 2048
    retq
```

} no control or
data **dependencies**
between these
code blocks

Data-Parallelism Through SIMD⁵ Instructions

`res[]` `col1[]` `col2[]`



- Read/compute/write 4 vector elements (of width 4×32 bits = 128 bits) at a time in **data-parallel** fashion.
- Relies on CPU's SIMD support (e.g., Intel® SSE registers `%xmm_i` and instruction `move double quad word`).

- **NB.** Requires care if
 - vectors `res[]` and `col1[]/col2[]` overlap in memory,
 - residual vector elements (see \emptyset) are to be processed.

⁵ SIMD: Single-Instruction, Multiple Data. Support is vendor-specific, rapidly evolving in contemporary CPUs. Intel's AVX2 registers: 512 bits, ARM NEON registers: 128 bits. Compilers implement CPU architecture flags.

Data-Parallelism Through SIMD Instructions (Vector Overlap)

C compiler `clang` (options `-O2 -fvectorize`) uses SIMD registers and instruction set.

- Extra prelude code checks for **vector overlap** on function entry. If so, jumps to non-vectorized (yet unrolled) version of code.
- Declare function arguments with modifier `__restrict__` to inform C compiler that vectors do not overlap (Δ omits all checks):

```
void PROJECT_sub_int_col_int_col(
    int *__restrict__ col1, int *__restrict__ col2, int *__restrict__ res)
{
    int i;
    :
}
```

Data-Parallelism Through SIMD Instructions (Core Loop)

Process 16 vector elements per iteration (SIMD + 2 loops unrolled; assumes no overlap of vectors `res[]` and `col1[]/col2[]`):

PROJECT_sub_int_col_int_col:

```

:
overlap check omitted
:
xorl %eax, %eax
loop:
    movdqu (%rdi,%rax,4), %xmm0
    movdqu 16(%rdi,%rax,4), %xmm1
    movdqu (%rsi,%rax,4), %xmm2
    psubd %xmm2, %xmm0
    movdqu 16(%rsi,%rax,4), %xmm2
    psubd %xmm2, %xmm1
    movdqu %xmm0, (%rdx,%rax,4)
    movdqu %xmm1, 16(%rdx,%rax,4)
    movdqu 32(%rdi,%rax,4), %xmm0
    movdqu 48(%rdi,%rax,4), %xmm1
    movdqu 32(%rsi,%rax,4), %xmm2
    psubd %xmm2, %xmm0
    movdqu 48(%rsi,%rax,4), %xmm2
    psubd %xmm2, %xmm1
    movdqu %xmm0, 32(%rdx,%rax,4)
    movdqu %xmm1, 48(%rdx,%rax,4)
    addq $16, %rax
    cmpq $2048, %rax
    jne loop
:

```

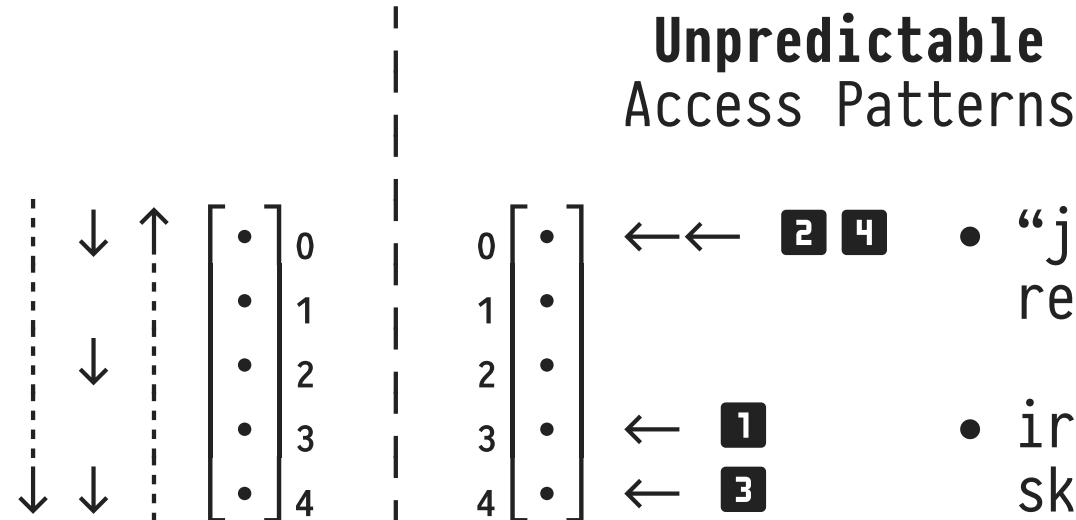
$4 \times 32 \text{ bits} = 128 \text{ bits wide}$

# %xmm0 \leftarrow_{128} col1[i+0...i+3]	loop #n
# %xmm1 \leftarrow_{128} col1[i+4...i+7]		
# %xmm2 \leftarrow_{128} col2[i+0...i+3]		
# %xmm0 \leftarrow_{128} %xmm0 - ₁₂₈ %xmm2		
# %xmm2 \leftarrow_{128} col2[i+4...i+7]		
# %xmm1 \leftarrow_{128} %xmm1 - ₁₂₈ %xmm2		
# res[i+0...i+3] \leftarrow_{128} %xmm0		
# res[i+4...i+7] \leftarrow_{128} %xmm1		
# %xmm0 \leftarrow_{128} col1[i+8 ...i+11]	loop #n+1
# %xmm1 \leftarrow_{128} col1[i+12...i+15]		
# %xmm2 \leftarrow_{128} col2[i+8...i+11]		
# %xmm0 \leftarrow_{128} %xmm0 - ₁₂₈ %xmm2		
# %xmm2 \leftarrow_{128} col2[i+12...i+15]		
# %xmm1 \leftarrow_{128} %xmm1 - ₁₂₈ %xmm2		
# res[i+8 ...i+11] \leftarrow_{128} %xmm0		
# res[i+12...i+15] \leftarrow_{128} %xmm1		
# }		
# } 2048 / 16 = 128 iterations		
# exit if %rax = 2048		
# (non-vectorized code not shown)		

6 | Predictable Memory Access and Prefetching

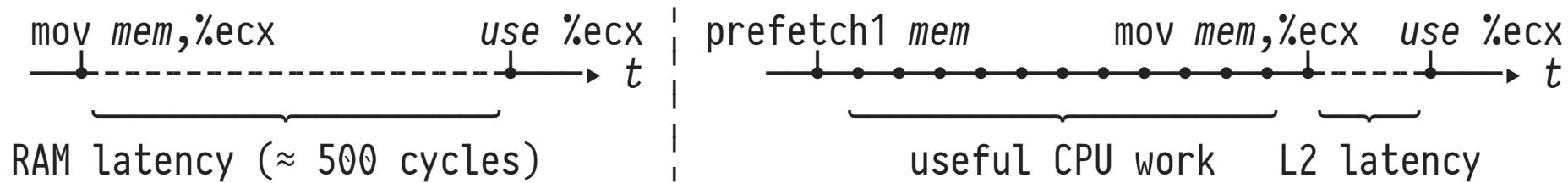
Predictable Access Patterns

- forward scans (possibly with skips)
- backward scans



- **Predictable:** CPU automatically issues **asynchronous memory prefetch** operations to preload caches and hide memory latency.
- **Unpredictable:** DBMS code adds explicit **software prefetch⁶ instructions** for memory addresses needed in the future:

#028



⁶ No-ops with side effect on the data cache. Example: `prefetcht1` loads into the L2 cache on Intel® Core i7.

7 | Implementing Selection in Tight Loops

The core of the **FILTER** operator is a **conditional statement** that is evaluated for each input vector element (⚠ simplified again):

```
int FILTER_lt_date_col_date_val(int* res, int* col, int val) #023
{
    int i, o = 0;

    for (i = 0; i < STANDARD_VECTOR_SIZE; i += 1) {
        if (col[i] < val) {                                // test filter condition (col < val)
            res[o] = i;                                     // build selection vector
            o += 1;
        }
    }

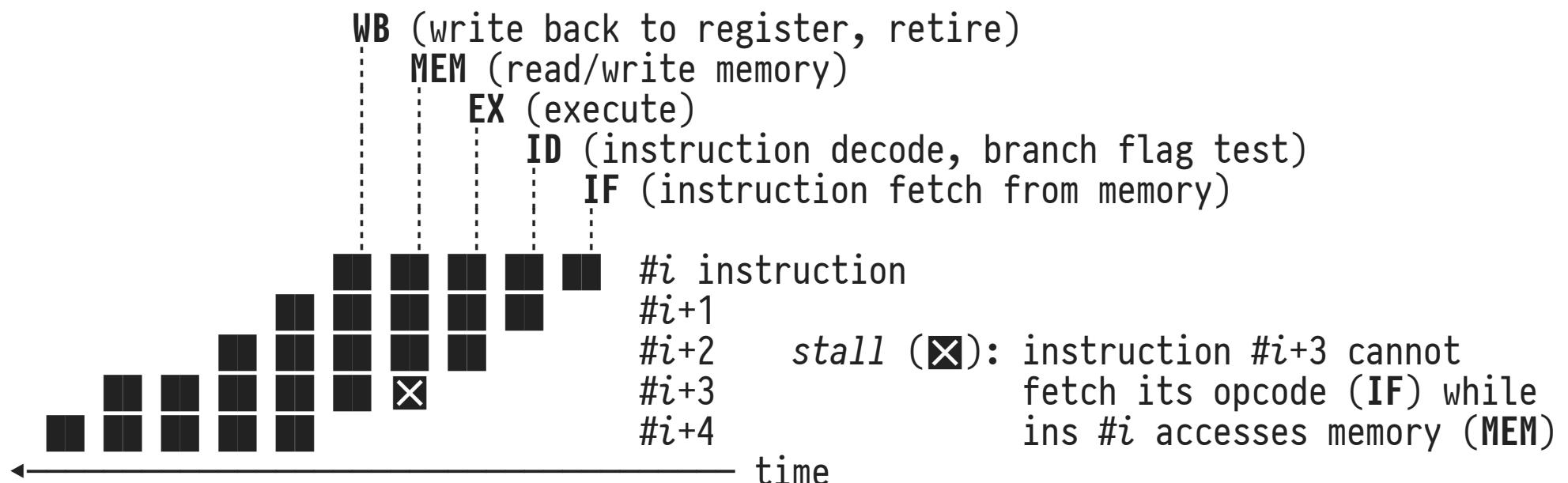
    return o;                                            // output cardinality
}
```

- Conditionals (and loops) lead to **branches** in generated machine code that disrupt the CPU's processing of straight-line instruction sequences. ⚡

Instruction Pipelining in Modern CPUs

Contemporary CPUs process instructions in a pipeline of execution stages that compete for CPU resources (e.g., the arithmetic logic unit or the memory bus).

- A simple five-stage pipeline:

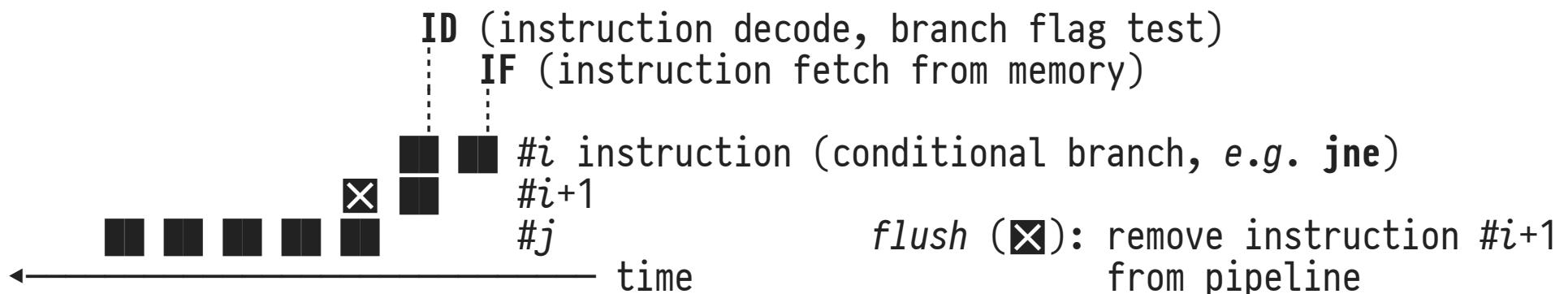


Branch Taken? Yes, Flush Pipeline

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

The simple pipeline decides the outcome of branch # i (at end of **ID**) only *after* instruction # $i+1$ has already been fetched (**IF**).

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

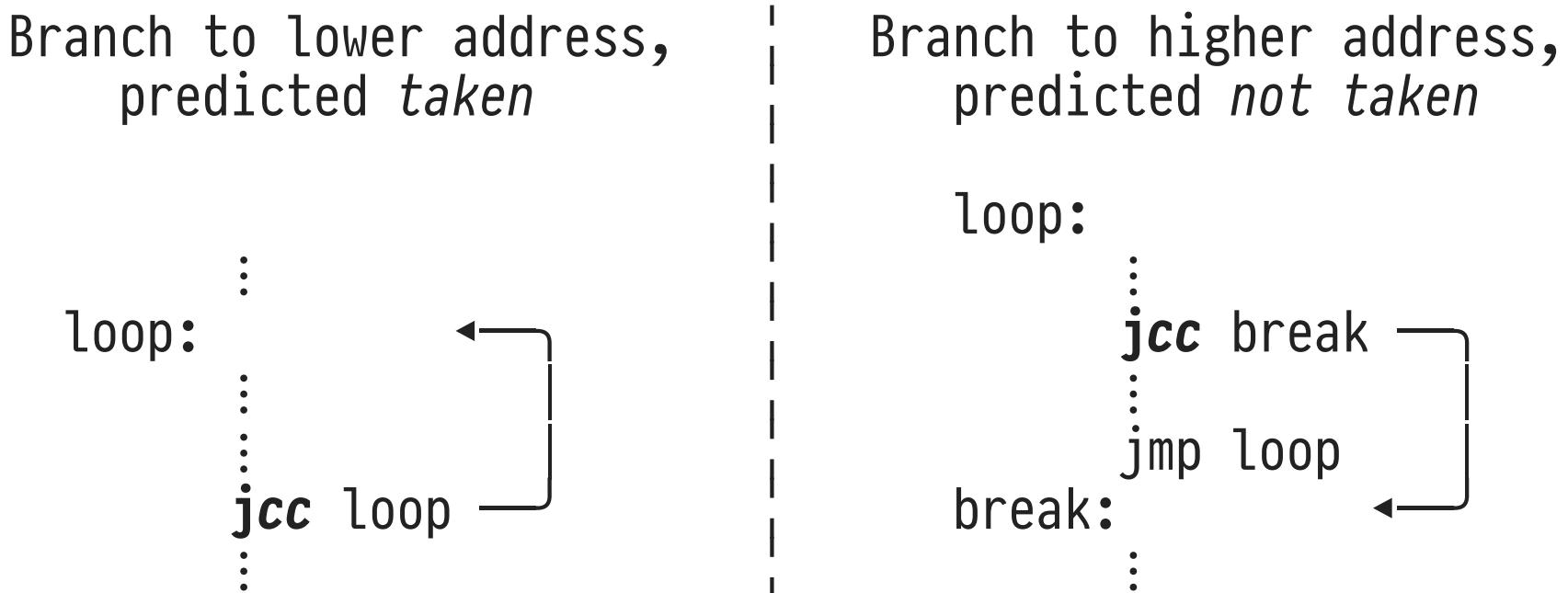


Branch Predictions: History and Heuristics

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

Branch prediction	Fetch instruction
<i>taken</i>	# <i>j</i>
<i>not taken</i>	# <i>i</i> +1

- Also: **heuristics** based on typical control flow patterns (instruction **jcc**: conditional branch on **condition code**):



Avoiding Branch Mispredictions

A **mispredicted branch** leads to

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

The resulting runtime penalty is significant (≈ 15 cycles).

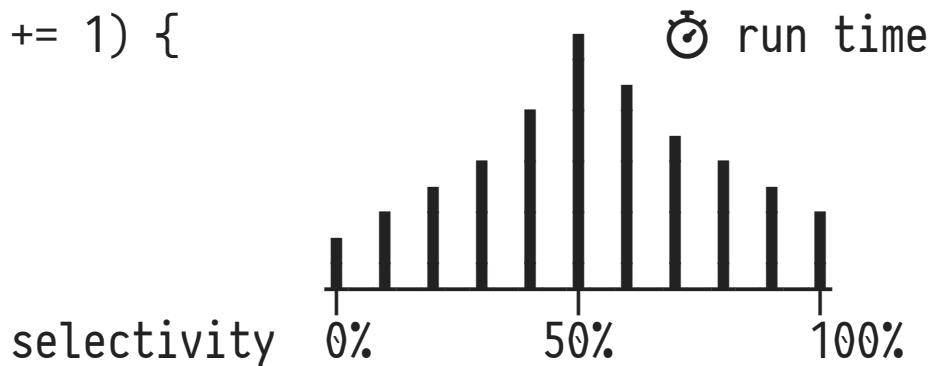
⌚ DBMS code aims to avoid branch mispredictions in tight loops:

- Prefer **branch-less** implementations of operator logic, or at least
- reduce the number of random/hard-to-predict branches.

Branch-Less Filtering

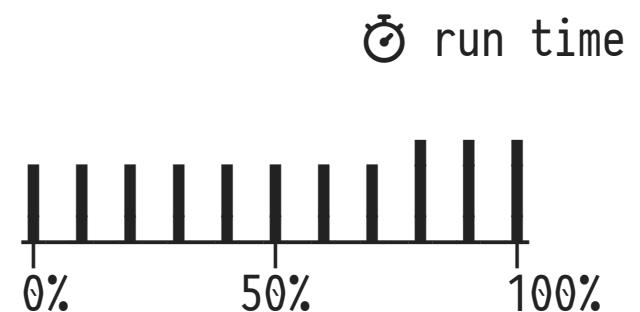
Measure wall-clock run time of core **FILTER** loop for different **selectivities** of predicate `col < val`:

① `for (i = 0; i < STANDARD_VECTOR_SIZE; i += 1) {
 if (col[i] < val) {
 res[0] = i;
 o += 1;
 }
}`



② `for (i = 0; i < STANDARD_VECTOR_SIZE; i += 1) {
 res[0] = i;
 o += (col[i] < v);
}`

$\equiv 1$ if predicate satisfied, else 0



②: 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 $p_1 \text{ AND } p_2$):

- Use branch-less selection via $o += p_1 \& p_2$ (NB. uses of C's bit-wise *and* operator $\&$).
- Identify the *more selective*⁷ (and thus more predictable) conjunct p_1 , say, then use

```
if (p1) {  
    res[0] = i;  
    o += (p2);  
}
```

(This approach particularly fits DBMSs that generate code from SQL queries—which DuckDB does *not* do.)

⁷ ⚠ This is important. Using `if (p2) ...` instead, where p_2 is unpredictable, immediately ruins the plan.