Hacking Postgres Executor For Performance

PGConf India 2025

Amit Langote

Microsoft

Speaker: Amit Langote

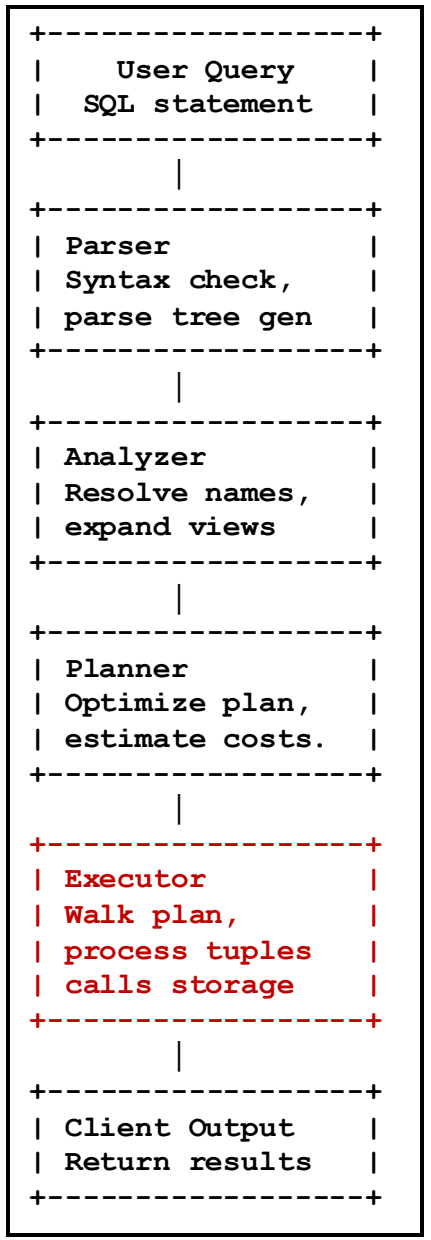
- Tokyo-based Postgres hacker, committer, major contributor
  - Mainly worked on adding table partitioning and then on improving its performance and scalability over the releases.
- Work at Microsoft
  - Previously EDB, NTT OSS Center

Agenda

- Context & Motivation
  - Why focus on executor performance and which aspect?
- CPU Performance & the Volcano Query Execution Model
  - CPU execution fundamentals: how modern CPUs execute programs efficiently
  - Postgres query execution model (Volcano) and its inefficiencies in CPU utilization
- Postgres Executor Optimizations *(Committed Work)*
  - Opcode-based expression evaluation, function call overhead reduction, more efficient reading of heap page tuples, more efficient tuple deforming, more efficient Scan node execution
- Larger Executor Rewrite & Future Directions
  - Opcode-based plan execution, batching and vectorization with SIMD, compiled plans (JIT)
- Discussion & Questions

Executor's Impact on Query Performance

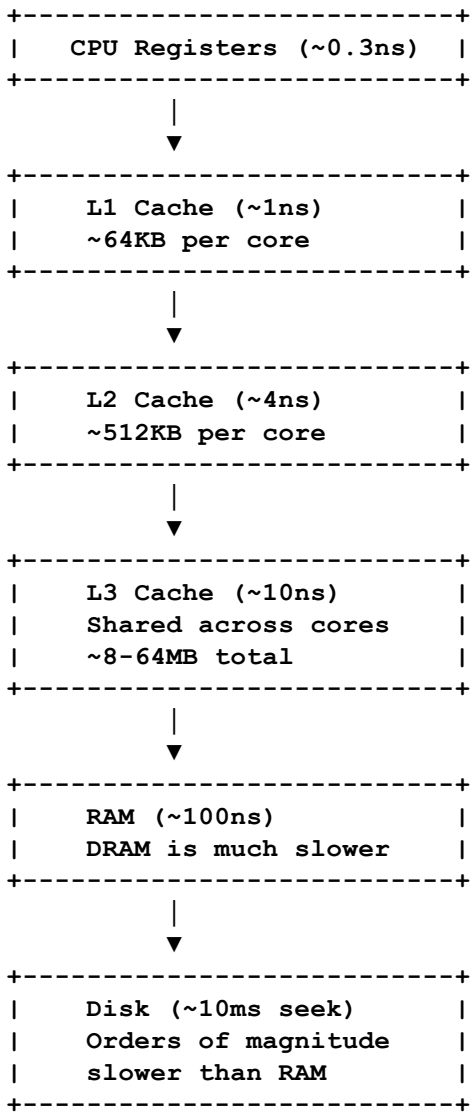
- OLTP (Transactional Queries):
  - Short queries, often returning few rows.
  - Executor startup time can be a significant fraction of total query time, especially with prepared statements or the extended query protocol.
- OLAP (Analytical Queries):
  - Large scans, aggregations—millions of tuples processed.
  - Inefficient CPU usage in the executor adds up over millions of tuples, increasing query execution time
  - Reducing per-tuple overheads can add up.



CPU Performance Primer

Cache Locality

- CPUs rely on fast caches
  - L1 ~1ns,
  - L2 ~4ns,
  - L3 ~10ns
  - RAM ~100ns+.
- Sequential access is efficient, scattered access is not.
- Cache misses stall execution while waiting for memory fetches.
- Frequent function calls can evict hot code from the instruction cache, increasing stalls.



Microsoft

Pipelined Execution

- Modern CPUs use pipelining to keep multiple instructions in flight.
- Pipeline stall = Cycles where the CPU cannot do useful operations.
- Common stall causes:
  - Cache misses → Waiting for slow memory access (~100ns+).
  - Branch misprediction → Flushing and restarting the pipeline.
  - Frequent function calls → Extra overhead managing return addresses and registers.
- Fewer stalls = higher throughput.

```
Cycle:   1   2   3   4   5   6   7   8
Instr1:  IF  ID  EX  WB
Instr2:      IF  ID  EX  WB
Instr3:          IF  ID  EX  WB
Instr4:              IF  ID  EX  WB
Instr5:                  IF  ID  EX  WB
```

```
Cycle:   1   2   3   4   5   6    7    8    9   10   11   12   13
Instr1:  IF  ID  EX  WB
Instr2:      IF  ID  EX  WB
Instr3:          IF  ID  EX  WB
Instr4:              IF  ID  EX*  --   --   --   EX  WB (Cache Miss)
Instr5:                  IF  ID   EX*  --   --   --  -- (Branch Misprediction)
Instr6:                                              IF  ID  EX  WB
```

Microsoft

Speculative Execution

- Speculative Execution keeps the CPU pipeline full by executing instructions before it is known whether they really need to be executed.
  - Branch Target Buffer (BTB) predicts jump targets to execute speculatively
  - Reorder Buffer (ROB) holds speculative results until confirmation.
  - Incorrect prediction → Pipeline is flushed, and ROB discards work.
- Pipeline stalls hurt performance when BTB fails, especially with indirect branches.

```
Cycle:    1   2   3   4   5   6   7   8
Instr1:   IF  ID  EX  WB
Instr2:       IF  ID  EX  WB
Branch:           IF  ID  EX  WB   (BTB Predicts Target!)
SpecInstr1:           IF  ID  EX  WB   (ROB Holds Result)
SpecInstr2:               IF  ID  EX  WB
```

```
Cycle:    1   2   3   4   5   6   7   8   9  10  11
Instr1:   IF  ID  EX  WB
Instr2:       IF  ID  EX  WB
Branch:           IF  ID  EX  WB   (Wrong Target Predicted!)
WrongInstr1:          IF  ID  EX  WB   (Speculative → Flushed!)
WrongInstr2:              IF  ID  --  --  (ROB Discards)
CorrectInstr1:                    IF  ID  EX  WB
CorrectInstr2:                        IF  ID  EX  WB
```

Microsoft

Branch Misprediction

- Branch mispredictions waste cycles by flushing the pipeline and reorder buffer.
- Primary causes of mispredictions:
  - Unpredictable (data-dependent) Branches
  - Indirect Branches (function pointers, virtual calls)
  - Return address misprediction (on RAS (Return Address Stack) overflow)
- Frequent function calls (especially indirect ones) increase mispredictions.
- Reducing mispredictions
  - Flattening execution paths to remove unnecessary branches.
  - Replacing indirect branches (function pointers, virtual calls) with direct calls where possible.
  - Loop unrolling reduces loop exit mispredictions.

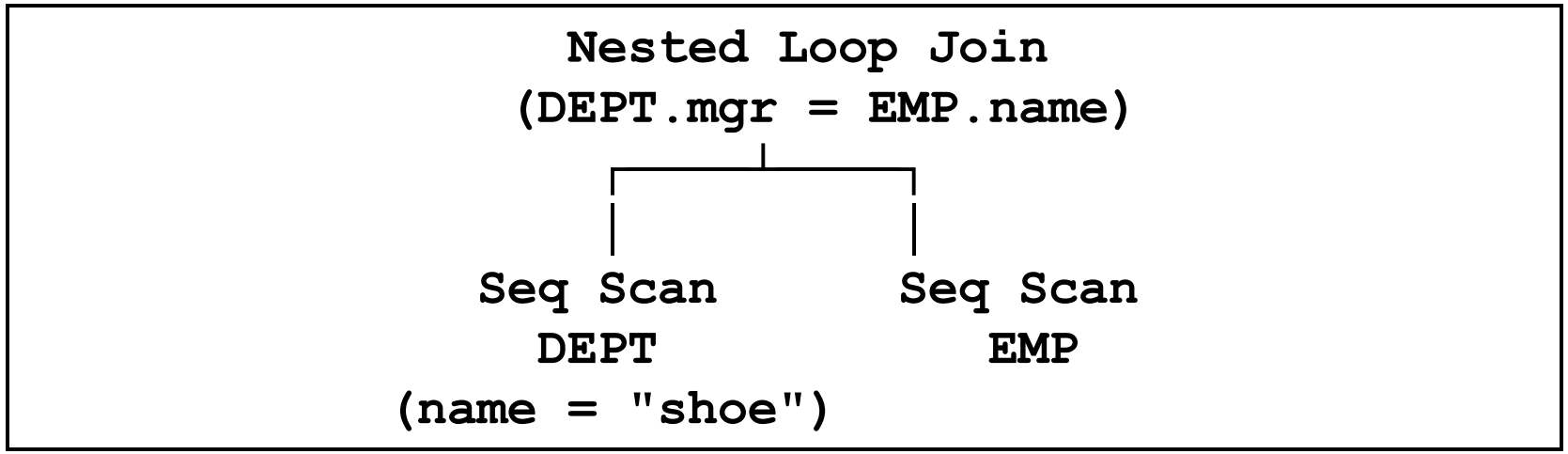
The Role of Compilers

- Compilers optimize code for efficient CPU execution:
  - Techniques such as inlining, loop unrolling, vectorization, and branch elimination
- Why doesn't this always work for Postgres?
  - Highly dynamic execution paths:
    - Queries aren’t known at compile time → No static optimization.
  - Function pointer-heavy design:
    - Execution functions are often called indirectly (ExecProcNode), limiting inlining.
  - Memory access patterns are unpredictable:
    - Query execution jumps across pages, limiting compiler optimizations.
- So, we need to hack Postgres executor for better CPU efficiency.

Postgres Execution Model

Volcano Model

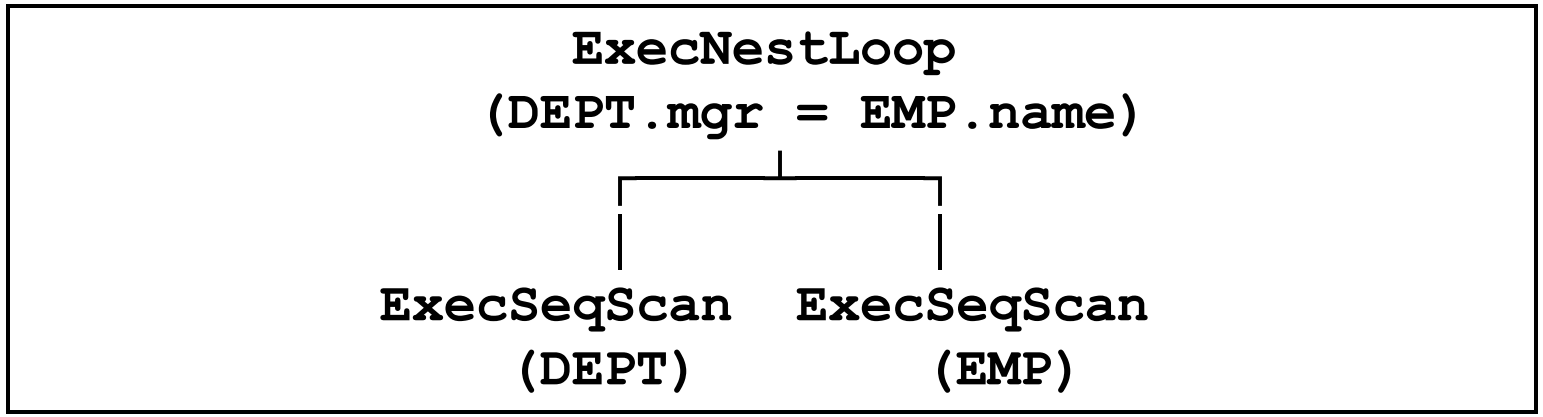
- A pull-based, top-down iterator model
- Query plans structured as a tree of operations (scans, joins, filters, aggregations)
- Each node requests tuples from its child nodes one at a time.



Microsoft

Volcano Model

- Iterator Model
  - Execution starts at the top of the query tree.
  - A parent node calls its child node's next() function to fetch a tuple.
  - This request propagates down the tree until a tuple is produced, then returns up.



Volcano Model: Issues

- Each tuple request requires multiple function calls up and down the plan tree.
- Scattered memory access prevents efficient CPU caching.
- Each tuple retrieval involves multiple condition checks, increasing mispredictions.

```
ExecNestLoop(Join)
  ├── ExecSeqScan(DEPT)
  │      └── SeqNext(ScanDesc) → Fetch tuple from DEPT
  ├── ExecSeqScan(EMP)  (for each DEPT row)
  │      └── SeqNext(ScanDesc) → Fetch tuple from EMP
```

Committed Optimizations for Executor Performance

Rewriting Expression Evaluation (v10)

- Problem
  - Tree-walking-based execution was Inefficient
  - Excessive function calls
  - Poor instruction cache efficiency

```
(a + 10) * b
```

```
ExecEvalExpr()
  ├── ExecEvalOp(*)                → Multiply
  │     ├── ExecEvalOp(+)          → Add
  │     │     ├── ExecEvalVar(a)    → Load column a
  │     │     ├── ExecEvalConst(10) → Load constant 10
  │     ├── ExecEvalVar(b)         → Load column b
```

Microsoft

Rewriting Expression Evaluation (v10)

- Solution
  - Replaced tree-walk with an opcode-based dispatch loop that uses a jump table.
  - No recursive function calls, improving branch predictability.
- Faster expression evaluation across all queries due to more efficient CPU execution

```
(a + 10) * b
```

```
ExecEvalExpr()
 ├── ExecEvalOp(*)                → Multiply
 │    ├── ExecEvalOp(+)           → Add
 │    │    ├── ExecEvalVar(a)     → Load column a
 │    │    └── ExecEvalConst(10)  → Load constant 10
 │    └── ExecEvalVar(b)          → Load column b
```

```
ExecInterpExpr()
-------------------------------------
EEOP_SCAN_VAR a       → Load column a
EEOP_CONST 10         → Load constant 10
EEOP_FUNCEXPR         → Add function (a + 10)
EEOP_SCAN_VAR b       → Load column b
EEOP_FUNCEXPR         → Multiply function ((a + 10) * b)
EEOP_DONE             → Finish evaluation
```

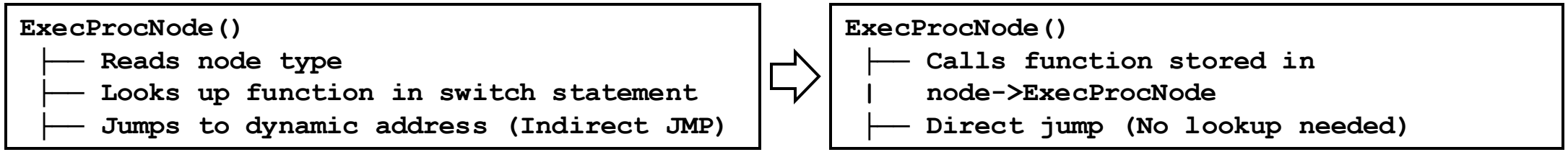
Reducing ExecProcNode() Overhead (v10)

- Problem
  - Centralized node dispatch function, ExecProcNode(), called for every plan node
- Overhead of extra function call per tuple and of indirect jumps due to function lookup based on node type

```
ExecProcNode()
  ├── Reads node type
  ├── Looks up function in switch statement
  ├── Jumps to dynamic address (Indirect JMP)
```

Reducing ExecProcNode() Overhead (v10)

- Solution
  - Each plan node now stores a function pointer
    - E.g., PlanState->ExecProcNode = ExecSeqScan
  - Avoids calling ExecProcNode() for every node, eliminating dispatch overhead.
- Removes per-tuple function call overhead.
- Reduces CPU stalls from indirect branch mispredictions.



Reducing Branches in heapgetpage() (v17)

- Problem:
  - Unnecessary per-tuple condition checks in heapgetpage()
  - Visibility and serialization conflict checks done for every tuple, even when conditions remained constant for a full page.
- Excessive branching, increasing mis-predictions and CPU stalls.

```
heapgetpage()
  ├── Lock buffer
  ├── Read page
  ├── for each tuple in page:
  │     ├── Check all_visible flag
  │     ├── Check snapshot visibility
  │     ├── Check for serializable conflicts
  │     ├── Store visible tuple
  ├── Unlock buffer
```

Microsoft

Reducing Branches in heapgetpage() (v17)

- Solution
  - Extracted the per-tuple loop into an inline function page_collect_tuple(), passing it precomputed all_visible and check_serializable once per page.
  - Compiler now generates specialized loops with fewer condition checks.
- Fewer unpredictable branches
- Sequential scans execute faster, especially on all-visible pages

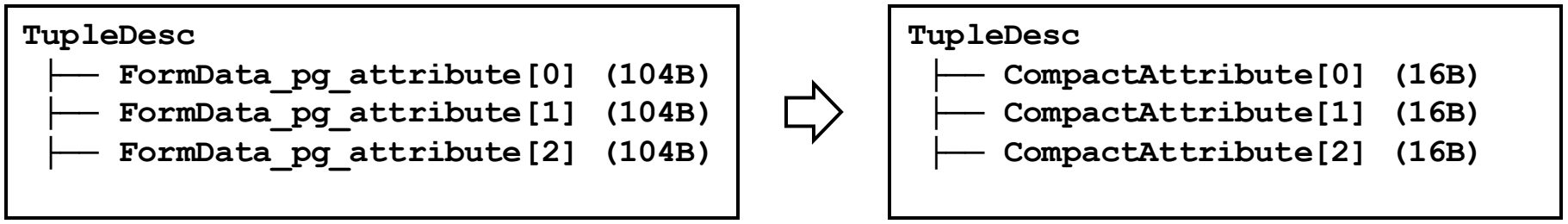
```
heapgetpage()
  ├── Lock buffer
  ├── Read page
  ├── for each tuple in page:
  │     ├── Check all_visible flag
  │     ├── Check snapshot visibility
  │     ├── Check for serializable conflicts
  │     ├── Store visible tuple
  ├── Unlock buffer
```

```
heapgetpage()
  ├── Lock buffer
  ├── Read page
  ├── Check all_visible flag once
  ├── Check serializable conflict flag once
  ├── Call page_collect_tuples() with precomputed flags
  │     ├── for each tuple in page:
  │     │     ├── Store visible tuple
  ├── Unlock buffer
```

Microsoft

Optimizing Tuple Deforming (v18)

- Problem
  - Tuple deforming accessed FormData_pg_attribute (104B per column) in TupleDesc, loading unnecessary cachelines.
- Solution
  - Replaced FormData_pg_attribute with CompactAttribute (16B per column) containing only essential fields, improving cache locality.



Microsoft

Optimizing Tuple Deforming (v18)

- Problem
  - Alignment logic used att_align_nominal()
  - Unpredictable branches per column.
- Solution
  - Switched from att_align_nominal() to TYPEALIGN(), eliminating branching in alignment calculations.

```
for each column:
    if (attalign == 'c')
        →  Align by 1 byte
    else if (attalign == 's')
        →  Align by 2 bytes
    else if (attalign == 'i')
        →  Align by 4 bytes
    else if (attalign == 'd')
        →  Align by 8 bytes
```

```
for each column:
        offset = TYPEALIGN(align, offset);
```

Microsoft

Optimizing Tuple Deforming (v18)

- Problem
  - slot_deform_heap_tuple() performed redundant NULL checks per column, even when no NULLs existed.
  - "Slow-path" logic was triggered for every attribute, even when not needed.
- Solution
  - Refactored slot_deform_heap_tuple() to use specialized loops, avoiding unnecessary NULL checks in common cases and enter the “slow” path only when needed.

```
for each column:
    if (HeapTupleHasNulls())
        check null bitmap
    compute attribute offset
    store attribute
```

```
if (no NULLs):
    for each column:
        compute attribute offset
        store attribute
else:
    for each column:
        check null bitmap
        compute attribute offset
        store attribute
```

Microsoft

Optimizing Tuple Deforming

- Fewer cache misses, fewer mis-predicted branches.
- 5-25% reduction in query time for large aggregation queries.

Refactoring ExecScan() for Inlining (v18)

- Problem
  - ExecScan() checked EvalPlanQual, quals, and projection on every call, even when not needed.
  - Frequent runtime condition checks led to branch mispredictions and instruction cache pressure.

```
ExecScan()
  ├── if (EvalPlanQual)
  │     ├── Fetch tuple for EPQ check
  │     ├── Apply EPQ logic
  │     ├── Return EPQ tuple
  ├── Fetch tuple (if not EPQ)
  ├── if (qual) → Evaluate WHERE clause
  ├── if (projection) → Apply projection
  ├── Return tuple
```

Microsoft

Refactoring ExecScan() for Inlining (v18)

- Solution
  - Moved core execution logic into an inline-able ExecScanExtended(), with parameters for EvalPlanQual state, qual, and projection states.
  - Specialized functions were introduced for different cases, avoiding unneeded checks.

```
ExecSeqScan()   (No qual, no projection)
  ├── Fetch tuple
  ├── Return tuple
```

```
ExecSeqScanWithQual()   (Qual, no projection)
  ├── Fetch tuple
  ├── if (qual) → Apply filter
  ├── Return tuple
```

```
ExecSeqScanWithProj()   (No qual, projection)
  ├── Fetch tuple
  ├── Apply projection
  ├── Return tuple
```

```
ExecSeqScanWithQualProj()   (Qual and projection)
  ├── Fetch tuple
  ├── if (qual) → Apply filter
  ├── Apply projection
  ├── Return tuple
```

Microsoft

Refactoring ExecScan() for Inlining (v18)

- Removes per-tuple branching where not needed.
- Up to 5% faster execution in SeqScan-heavy queries.

Larger Executor Rewrite: Future Directions

Opcode-Based Execution

- Loop over compact operations, no recursive function calls.
- Fewer function calls, fewer branches.
- Avoids deep call stacks.

```
0: init_sort
1: seqscan_first
2: seqscan [j empty 5] > s0
3: qual [j fail 2] < scan s0
4: hashagg_tuple [j 2] < s0
5: drain_hashagg [j empty 7] > s1
6: sort_tuple [j 5] < s1
7: sort
8: drain_sort [j empty 10] > s2
9: return < s2 [next 8]
10: done
```

```
for each tuple:
    execute_opcode(OP_FILTER)
    execute_opcode(OP_PROJECT)
    execute_opcode(OP_AGGREGATE)
    ..
```

Redesigning the executor (async, JIT, memory efficiency) (Andres Freund, 2018)

Microsoft

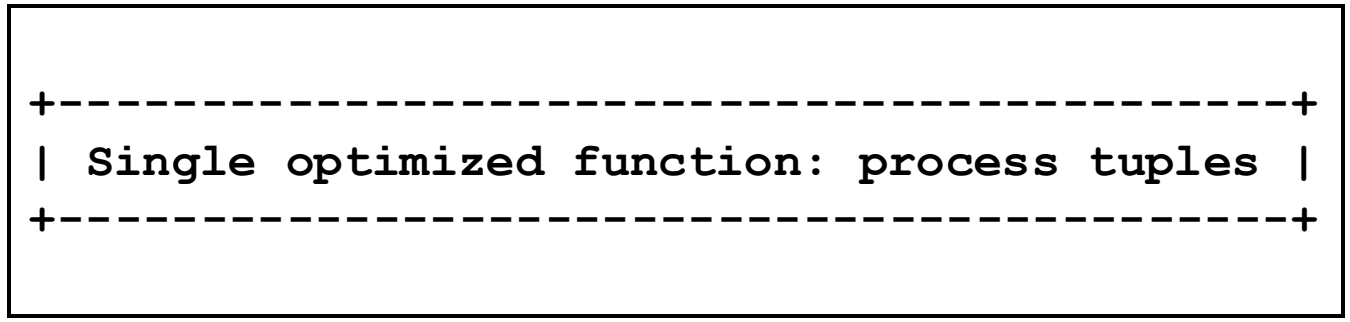
Batch Processing & Vectorized Execution

- Process batches of tuples per function call, reducing per-tuple function call overhead.
- Leverage SIMD if storage is columnar, enables efficient vectorized operations.
- Fewer next() calls, fewer CPU stalls, keeps pipeline full, reduces branch mispredictions.

```
for each batch:
    apply_filter(batch)
    apply_projection(batch)
    apply_aggregation(batch)
```

Compiled Execution (JIT Compilation)

- Generate optimized machine code, no interpretation overhead.
- Eliminates per-tuple ExecProcNode() calls.
- No function pointer dispatch, fewer mispredictions.
- Upfront compilation cost → Best for long-running queries.



Microsoft

Thank you. Questions?

Appendix: Committed Patches

- https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=b8d7f053c5c2bf2a7e8734fe3327f6a8bc711755
  - Faster expression evaluation and targetlist projection (Andres Freund)
- https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=cc9f08b6b813e30789100b6b34110d8be1090ba0
  - Move ExecProcNode from dispatch to function pointer-based model (Andres Freund)
- https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=a97bbe1f1df9eba0b18207c321c67de80b33db16
  - Reduce branches in heapgetpage()'s per-tuple loop (Andres Freund)
- https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=5983a4cffc31640fda6643f10146a5b72b203eaa
  - Introduce CompactAttribute array in TupleDesc (David Rowley)
- https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=db448ce5ad36a2754e4e75900b180260143aacf8
  - Optimize alignment calculations in tuple form/deform (David Rowley)
- https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=58a359e585d0281ecab4d34cab9869e7eb4e4ca3
  - Speedup tuple deformation with additional function inlining (David Rowley)
- https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=fb9f955025f7609fd3da0d7e33b77438ddc765de
  - Refactor ExecScan() to allow inlining of its core logic (Amit Langote, David Rowley)