FOSDEM PGDay 2026 - Brussels

Batching in Executor

Toward batch-mode execution in Postgres

Amit Langote, Postgres Committer

Previously...

- PGConf India 2025, POSETTE 2025: "Hacking Postgres Executor for Performance”
- "Batching could be the foundation for efficient OLAP execution in Postgres”
  - Amortize per-tuple overhead across many tuples
  - Path toward vectorized execution
- So I went and tried to build it.

Agenda

1. Why batching matters - CPU overhead in row-at-a-time
2. The Volcano model and its limits
3. Batching approaches - selective vs batch-native
4. One approach: prototype and results
5. Open questions and future work

1

Why Batching

Per-Tuple Overhead on Modern CPUs

- Indirect Call Overhead
  - node->ExecProcNode(node) target changes as execution bounces between nodes, table AM callbacks add more indirection
- Data-Dependent Branches
  - Qual pass/fail depends on tuple data - CPU mispredicts ~50% of the time at 50% selectivity
- Cache Inefficiency
  - Bouncing between Agg code, scan code, heap AM code pollutes L1 instruction cache

2

The Volcano Model

The Volcano Iterator Model

- Postgres's executor processes one tuple at a time - a design inherited from the classic iterator model that favors modularity but adds significant per-tuple overhead.

```
TupleTableSlot *ExecProcNode(node)
{
    slot = ExecProcNode(child);
    <do things>
    return slot;
}
```

- Strengths:
  - Modular: nodes compose freely. Simple: each node is self-contained. Memory-efficient: one tuple in flight.

The Volcano Iterator Model Bottleneck

- The iterator model remains a bottleneck for analytic workloads, even as I/O has gotten faster.
- OLTP Workloads
  - ~100 tuples per query. I/O and locking dominate. Volcano works well.
- OLAP Workloads
  - 10M+ tuples per query. Data in memory or on fast NVMe. CPU becomes the limiting factor.
- Per-tuple overhead limits instruction and cache efficiency even in simple scans.

Recent Improvements

- Recent improvements have reduced overhead in key paths, but the iterator model remains a bottleneck:
  - Opcode-based expression evaluation (v10+)
  - JIT compilation (v11+)
  - Read streams for async I/O (v17+)
  - Faster tuple deforming (v18)
  - Scan inlining (v18)
- These reduce per-tuple cost, but we still pay it for every tuple.

3

What is Batching

What is Batching?

- Process multiple tuples together instead of one at a time. Amortize fixed costs across the batch.

```
// Row-at-a-time
for each tuple
{
    call_overhead();
    process();
}
```

```
// Batched
call_overhead();
for each tuple
{
    process();
}
```

Batching Already Exists in Postgres

- The storage layer already thinks in batches. It is the executor that is behind:
  - Heap Pages
    - Each 8KB page holds ~40-100 tuples. We already fetch pages, not individual tuples.
  - Index Leaf Pages
    - B-tree leaves contain many TIDs. Index prefetching uses leaf pages as batches.
  - COPY / Bulk Insert
    - Multi-insert, batched WAL writes. Already batch-optimized.
- So, storage returns batches, but executor unwraps them one tuple at a time

Two Approaches to Batching

- Option A: Batch-Native Executor
  - Build separate vectorized executor. All nodes work on column batches. Think DuckDB, Velox.
  - Pros: maximum gains.
  - Cons: years of work, two executors to maintain.
- Option B: Selective Batching (this talk)
  - Extend existing executor incrementally. Add batch mode to nodes that benefit most. Preserve row semantics.
  - Pros: incremental changes, one codebase.
  - Cons: smaller gains than full vectorization.

PGConf.dev 2025 Unconference: 3 Approaches

- Community discussion identified three possible directions (not mutually exclusive):
  - Approach 1: Batching Inside SeqScan
    - Loop inside ExecSeqScan fetches multiple tuples before returning. Low disruption.
  - Approach 2: Specialized Executor
    - Separate executor for batch-friendly patterns (SeqScan to Agg). Medium disruption.
  - Approach 3: General Batch Infrastructure
    - ExecProcNodeBatch() + TupleBatch abstraction. Extensible to all nodes.
    - This patch: Combines 1 and 3. Focused on SeqScan, introduces TupleBatch.

4

One Approach

Design, Implementation, and Results

Core Abstractions

- New batch-oriented TableAmRoutine callbacks

```
+/* ------------------------------------------------------------------------
+ * Batched scan support
+ * ------------------------------------------------------------------------
+ */
+
+ void  *(*scan_begin_batch)(TableScanDesc sscan, int maxitems);
+ int    (*scan_getnextbatch)(TableScanDesc sscan, void *am_batch,
+                             ScanDirection dir);
+ void   (*scan_end_batch)(TableScanDesc sscan, void *am_batch);
```

Core Abstractions

- TupleBatch
  - Keeps data in native format, supports columnar access, enables ops like count += ntuples.

```
struct TupleBatch
{
    void *am_payload;    // HeapBatch, etc.
    TupleBatchOps *ops;  // heapam_materialize_all(), etc.
    int ntuples;
    int max_tuples;
}
```

Core Abstractions

- New batched node execution function

  ```
  TupleBatch *ExecProcNodeBatch(PlanState *node)
  ```

  - For example, SeqScanBatch()

Batched Scan Execution Flow

Per-tuple (current)

```
ExecProcNode()
 ExecSeqScan()
  ExecScanExtended() return TupleTableSlot
   SeqNext()
    table_scan_getnextslot()
     heap_getnextslot()
      heapgettup_pagemode()
```

× 10M times

Per-batch (with patch)

```
ExecProcNode()
 ExecSeqScan()
  ExecScanExtendedBatch() returns TupleBatch
   SeqNextBatch()
    table_scan_getnextbatch()
     heap_getnextbatch()
      heapgettup_pagemode_batch()
```

× 156K times (for 10M rows)

Batched Qual Evaluation

- Adapt expression evaluation to process WHERE clauses across batches:
  - Old: per-tuple ExecQual()

```
for each tuple:
    result = ExecQual(qual, slot)
```

  - New: ExecQualBatch() with bitmask
    - New batch-aware ExprEvalOps for 2-arg OpExpr and NullTest

```
results_bitmask = ExecQualBatch(qual, batch)
```

- Constraints:
  - Only simple AND-trees of supported expressions
  - Only leakproof operators (security barrier safety)
  - Falls back to per-tuple for more complex quals

Batched Agg Execution Flow

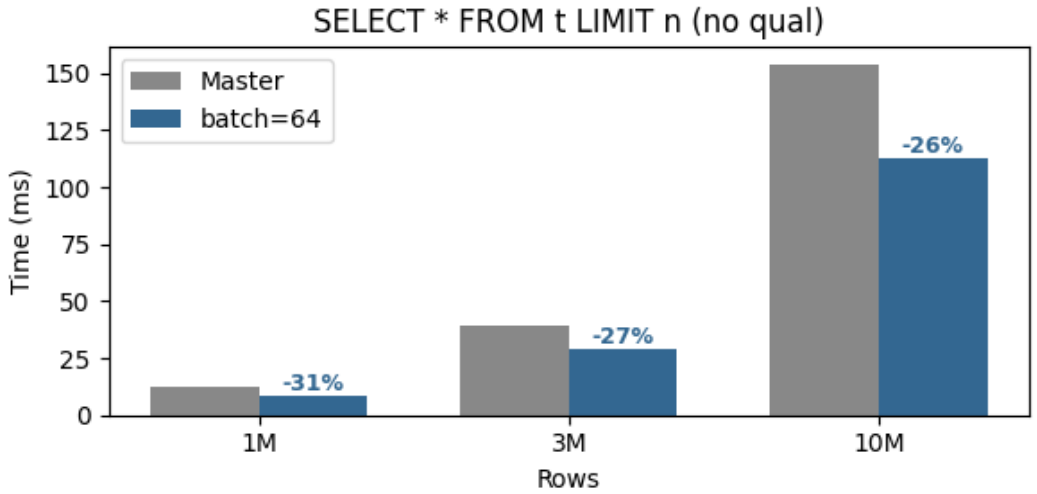
- Example: SELECT count(*) FROM t

1. Agg calls ExecProcNodeBatch(SeqScan)
2. SeqScan calls scan_getnextbatch(heap, batch, 64)
3. Heap TAM fills batch with ~40 tuples from current page
4. SeqScan returns batch to Agg
5. Agg processes all tuples in tight loop or just does
   count += batch->ntuples
6. Repeat until scan exhausted

- Before: Agg calls SeqScan 10M times, SeqScan calls heap 10M times
- After: Agg calls SeqScan ~156K times, SeqScan calls heap ~156K times

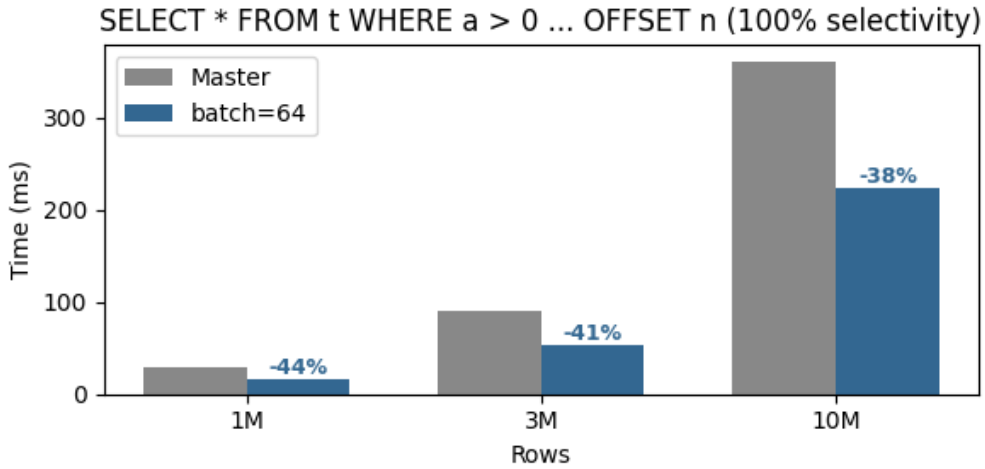
Microbenchmark Results: Batched Scan

- Fully cached, batch size 64. Comparing master vs patched:
https://postgr.es/m/CA+HiwqH-2GmTKLW9kHdnqV4KdFiPfuAdVK2TgqOM2JaaeUYXnw@mail.gmail.com



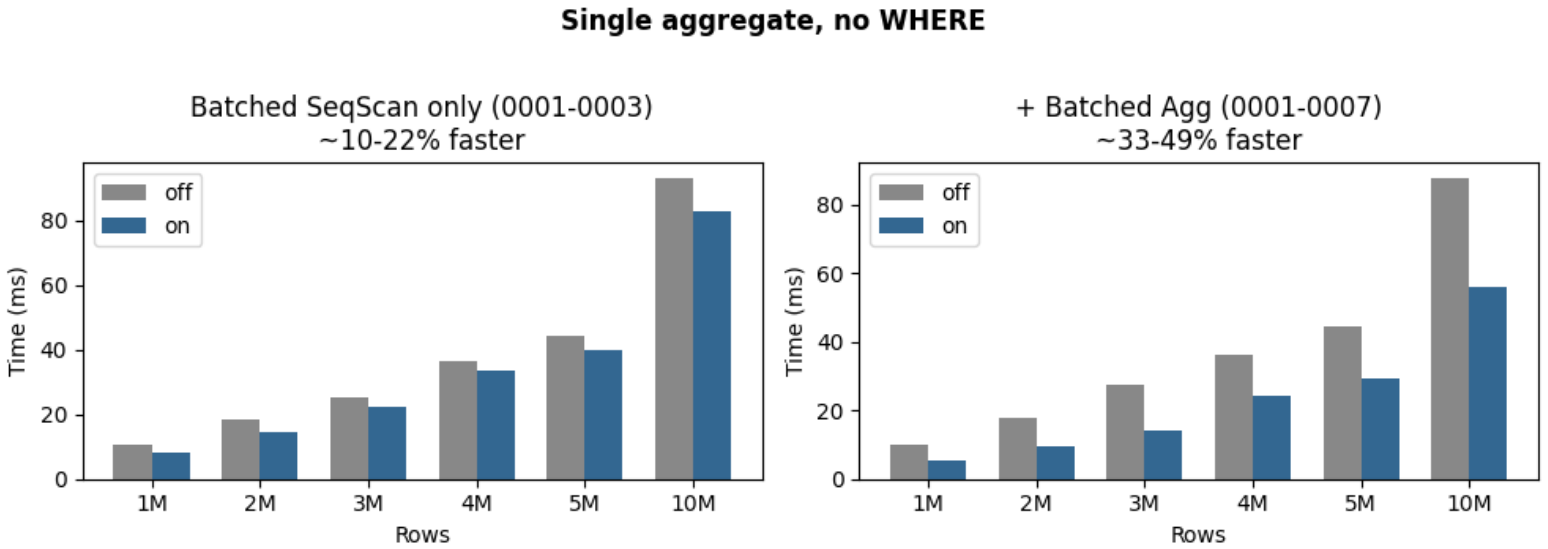
Microbenchmark Results: Batched Scan + Qual

- Fully cached, batch size 64. Comparing master vs patched:
https://postgr.es/m/CA+HiwqH-2GmTKLW9kHdnqV4KdFiPfuAdVK2TgqOM2JaaeUYXnw@mail.gmail.com



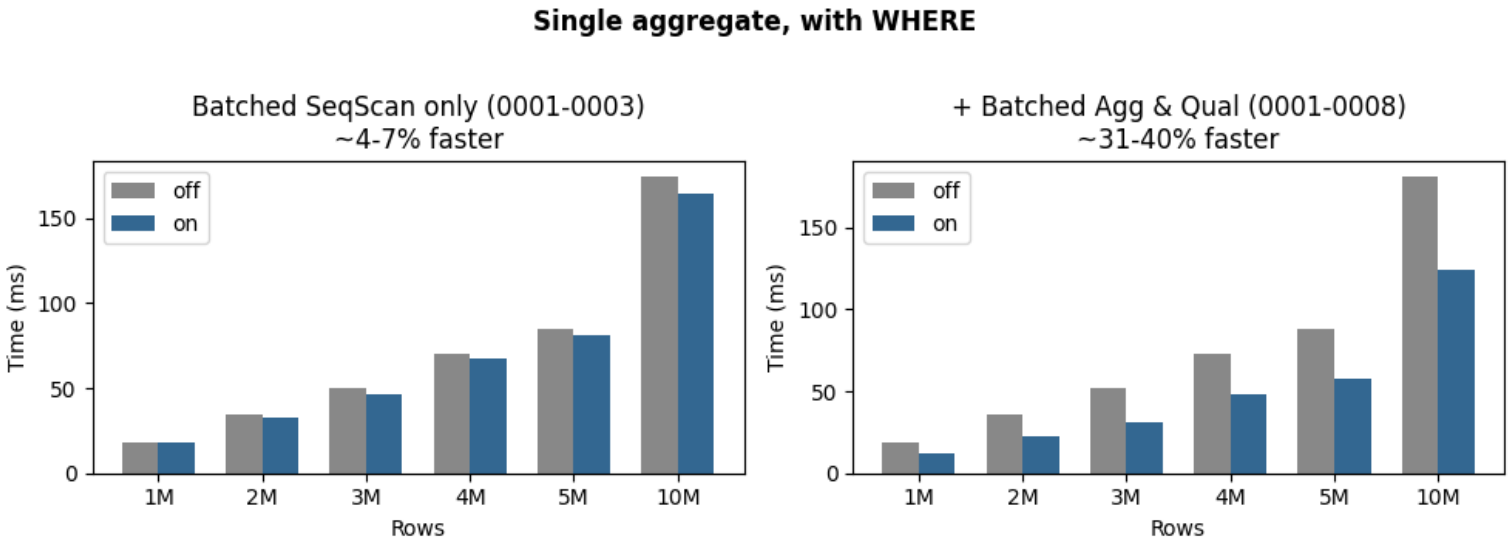
Microbenchmark Results: Batched Scan + Agg

- Fully cached, batch size 64. Comparing master vs patched:
https://postgr.es/m/CA+HiwqFfAY_ZFqN8wcAEMw71T9hM_kA8UtyHaZZEZtuT3UyogA@mail.gmail.com



Microbenchmark Results: Batched Scan + Qual + Agg

- Fully cached, batch size 64. Comparing master vs patched:
https://postgr.es/m/CA+HiwqFfAY_ZFqN8wcAEMw71T9hM_kA8UtyHaZZEZtuT3UyogA@mail.gmail.com



Patch Status

- Target for v19
  - Table AM batch API + heapam: HeapBatch, scan_begin / getnext / end_batch
  - SeqScan batching + TupleBatch: inslots[], materialize_all, executor_batch_rows GUC
- In development (shows potential)
  - Batched qual evaluation: new EEOPs, ExecQualBatch(), separate interpreter (ExecInterpQualBatch()), WIP.
- Future work
  - ExecProcNodeBatch(): node interface returning batches
  - Aggregate batching: use ExecProcNodeBatch(), batched agg transitions

Hard Questions

- Optimal Batch Size
  - Larger batches = better amortization but more memory. 64 tuples × many columns × deep plans = pressure on L2/L3. Currently a GUC; needs tuning.
- Optimizer Involvement
  - Should planner decide batch vs row mode? Cost model changes? For now: executor decides at runtime based on node capabilities.
- LIMIT Queries
  - LIMIT 1 with batch size 64 = wasted work. Solution: adaptive batch size ramp-up.
- Mode Mixing
  - Parent expects rows, child produces batches? Need adapter logic or graceful fallback.
- No Regressions
  - Must not slow OLTP. Batching is opt-in per node; nodes that don't benefit simply don't implement it.
  - The ability to turn batching off with zero overhead of the new code.

Future Work

- Near Term
  - Add batch support to other Scan nodes, Hash/Sort, and TPC-H benchmarks
- Medium Term
  - Columnar TAM integration (late materialization), SIMD vectorization, batch-native functions
- Longer Term
  - Batch-aware joins (HashJoin probe batching), projection batching, planner cost model for batch mode
- Long Term
  - Aggregate batching via ExecProcNodeBatch() - Agg pulls batches from child, batched transitions for sum/count/avg

Enable Postgres to compete on analytics while preserving OLTP strengths

Key Takeaways

- The iterator model remains a bottleneck for analytic workloads - per-tuple overhead limits efficiency even in simple scans
- This prototype enables executor nodes to operate on batches of tuples instead of individual slots
- ExecProcNodeBatch() API and TupleBatch abstraction preserve Postgres's row-based semantics and plan structure
- Early results show meaningful improvements, paving the way for broader batch-aware execution

Thank You!

Questions and Discussion

Patch Thread: pgsql-hackers "**Batching in executor**"

Thanks to Andres Freund, David Rowley, Tomas Vondra, Peter Geoghegan, and everyone who provided feedback

POSETTE: An Event for Postgres 2026—in its 5th year

- Free & virtual developer event
- Organized by PG team @ Microsoft
- Jun 16-18, 2026
- CFP is open until Sun Feb 1st @ 11:59pm PST

PosetteConf.com/2026/cfp