Pushing the Limits of the Index API

Building a Columnar Store without a TAM

Stu Hood - 05/21/2026

1. The Goal

Context: What is ParadeDB?

- An Elasticsearch alternative built as a Postgres extension
  - Full featured, fast, easy to use (no ETL)
  - In Rust, using `pgrx`, Tantivy, and DataFusion
- Built as an IAM with Custom Scans.

Context: Search?

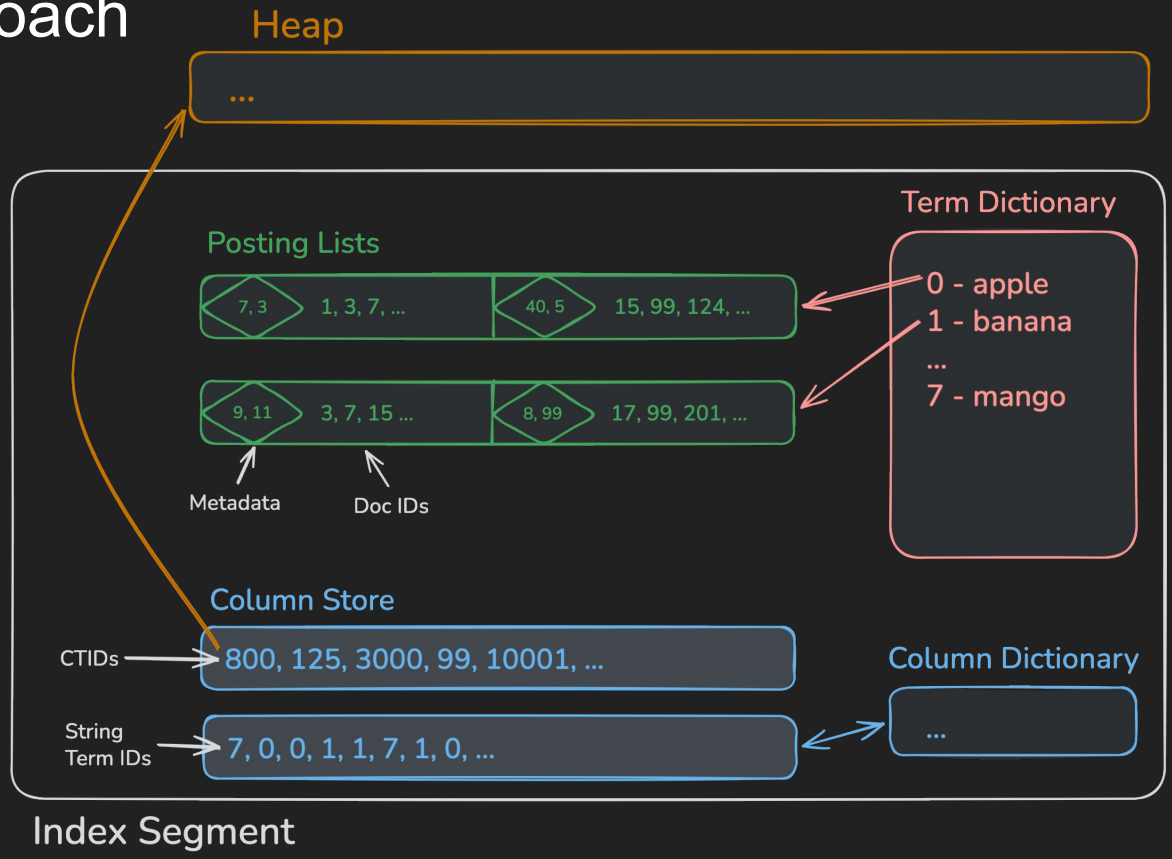
- Focused on full-text search over OLTP schemas
  - “Relational search”
  - Single table proven, joins in beta.
- Search and OLAP techniques for:
  - Low-latency Top-K
  - Search aggregates (”facets”)
  - Example!

Why IAM instead of TAM?

- Be Postgres “Native”!
  - No disruption for existing OLTP use cases.
  - Heap is battle tested; building a competitive TAM is a high bar.
- Without also changing IAM impls, must still quack like Heap.
  - c.f. OrioleDB’s bridge indexes
- No ability to push down predicates in a TAM.
  - We execute almost all quals with the index, the rest are heap lookups.
  - So either way, lots of Custom Scans.

Multi-Layout IAM Approach

- Two primary index structures:
  - Posting lists.
  - Column store.
- Usual execution path:
  - Execute posting list intersections for equality filters
  - Scan column store for range filters (usually)
  - Project from column store if covering
- Multiple immutable Segments: parallel workers at the Segment level



Schema

```
CREATE INDEX search_idx ON items
USING bm25 (
  id,
  (description::pdb.icu),
  category
)
WITH (key_field='id');
```

```
CREATE TYPE item_fields AS (
    description pdb.simple('stemmer=english'),
    category pdb.literal,
    in_stock BOOLEAN
);

CREATE INDEX search_idx ON mock_items
USING bm25 (id, (ROW(description, category, in_stock)::item_fields))
WITH (key_field='id');
```

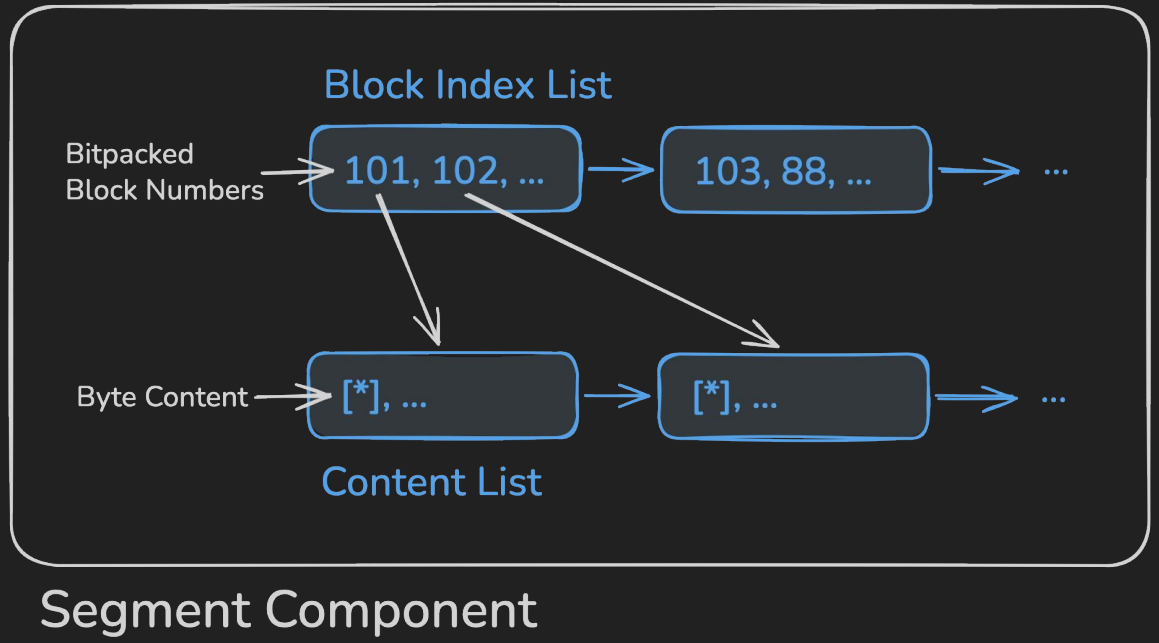
Why Multi-Column?

- ParadeDB is massively multi-column!
- Allows for offset-based (“DocId”) intersection of predicates
  - Single-relation Custom Scans take over 98% of quals to execute with the index
  - Key strength of search indexes is full fidelity* bitmap intersection
    - * As opposed to Bitmap Index Scan’s block-level fidelity
  - `tid` stored once per tuple
- Predicates applied based on `selectivity * cost` (per-Segment), seeking forward on the index when sparse
- tl;dr: Don’t choose between indexes!
  - Avoid the prefiltering problem.

2. Storage

Index Mapping to Blocks

- Segments have 6+ “Components”
  - Posting lists, columns, term frequencies, vectors (soon!)
- Component mapped to Postgres blocks using simple linked lists
  - Indexed for random access
- Specialized component structures have not been worth it (yet!)
- Uses generic WAL records



Segment Component

Buffer Manager Gymnastics

- Tantivy byte range APIs:
  - `FileSlice` - a thin handle to a range of bytes in a Segment Component
  - `OwnedBytes` - a loaded slice of bytes acquired from storage
- Tantivy default: MMAP
  - Fully random access, larger than memory slices of memory
  - Naturally lead to some lax handling of `FileSlice`/`OwnedByte` boundaries
- Zero-copy reads
  - Segments are immutable
  - Pin unlocked Postgres buffers to back `OwnedBytes`
  - Zero-copy except when read spans a block boundary

MVCC

- Immutable layouts are incredibly dense for scans!
- But what about deletes?
  - Segments have a “delete vector/bitmap” Component
  - IAM hooks `amvacuumcleanup` and `ambulkdelete` batch-recreate the delete bitmaps during VACUUM
    - LSM trees are very amenable to batching!
  - Compaction/merging of Segments prunes deleted data
    - In background workers, or foreground for backpressure
- Index access time visibility pruning via visibility map, ideally
  - More on this later!

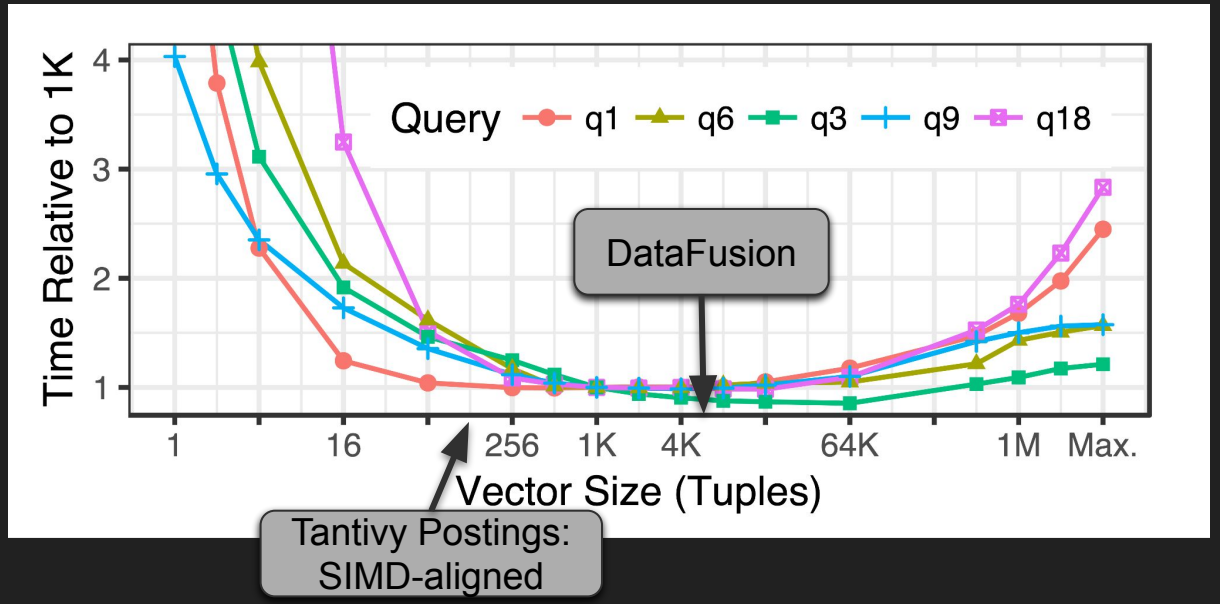
3. Proven and Frontier

Proven: Scans & Aggregates

- High-mileage Custom Scan hooks:
  - `set_rel_pathlist_hook` - single-table Top-K, columnar, or “normal” scans
    - Tantivy only
  - `create_upper_paths_hook` - aggregates
    - Tantivy for smaller-than-memory, DataFusion for larger-than-memory
- Leverage Tantivy’s core strengths:
  - Fundamental: Full-fidelity predicate intersection
  - Vectorization
  - Late materialization
  - Dynamic filters
  - Predicate pullup

Proven: Vectorization

- Execution must balance function call overhead (left) against L1/L2 cache thrashing (right).
- Posting list intersections biased left.
  - SIMD-aligned 128 row blocks.
- Dense analytic scans and aggregates bias right.
  - …post filtering.



Żukowski, Marcin - Balancing vectorized query execution with bandwidth-optimized storage

Kersten, Timo, et al. - Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask

Proven: SIMD-aligned Pruning

- Block-Max WAND: Top-K on score.
  - Evaluate a block's metadata (max inputs to BM25 score).
  - If the block's max score cannot exceed the current threshold (a dynamic filter from Top-K), skip the block without decompression.
- Blocks are sized to be SIMD-register-size-aligned, AND to allow skipping past blocks for sparse intersections.
  - 128 * 32 bit doc ids per block
  - SIMD-BP128: 32 ops * 128 bit registers

Ding, Shuai, and Torsten Suel. - Faster top-k document retrieval using block-max indexes

Lemire, Daniel. - Fast integer compression: decoding billions of integers per second

Proven: Vectorization: Aggregates

- After filtering using the index (sparse: 64-128 rows at a time), aggregations accumulate 2048 rows before executing (dense).
- No UDFs at this level: all operators vectorized.

Proven: Late Materialization

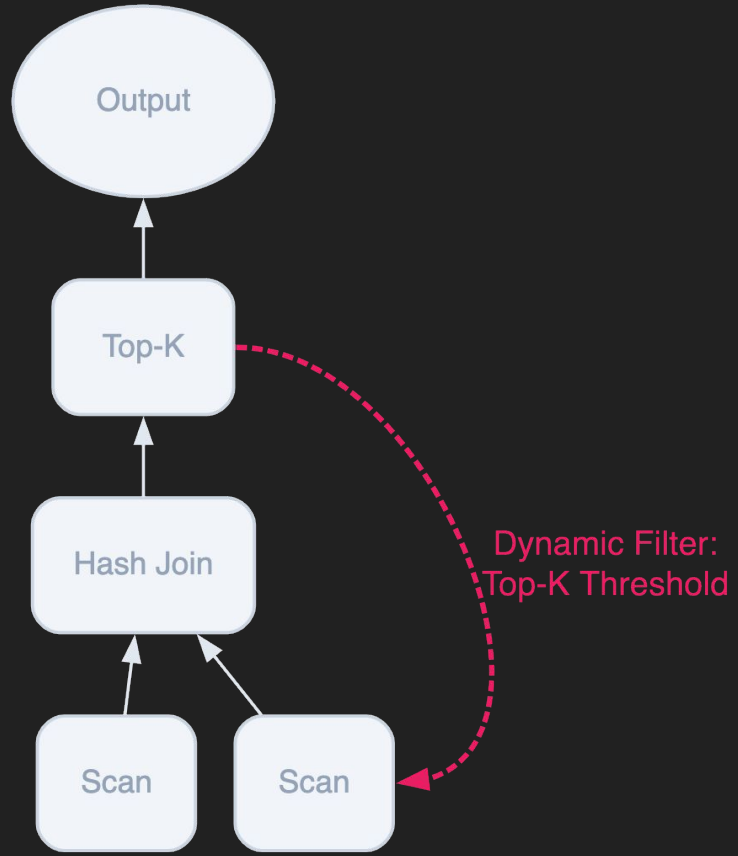
- Fetching column values “as late as is reasonable” during a query.
  - If:
    - 1. file format gives reasonable random access to columns
    - 2. selective filters can be applied first
    - 3. plan nodes haven't shuffled the access order too much
  - …then avoids IO, reduces memory bandwidth
- "Late" is a spectrum:
  - **Top:** At the target list.
    - Smallest possible set of rows, but ~random.
  - **Middle:** Late application of predicates (due to predicate pullup).
    - Primarily relevant during joins.
  - **Bottom:** Filters/masks applied progressively to columns during a scan.
- More on this later!

Proven: Late Materialization for Top-K

- Late materialization critical for Top-K:
  - For a Top-K on score, fetch *no* column or heap data (just postings).
  - For a Top-K on a column, fetch exactly one column (and postings).
- After computing Top-K:
  - Visibility filter and fetch `tid` (see Predicate Pullup!)
  - Go to heap for target list of the surviving K.

Proven: Dynamic Filters

- Passing information “down” a plan, while tuples flow “up”.
  - Ideally all the way to the bottom of the plan, to drive late materialization.
- Examples:
  - Top-K’s threshold value for Block Max WAND (critical!)
  - HashJoin’s entire build-side
  - SortMergeJoin’s per-side thresholds
- Evolved independently in OLAP and search:
  - OLAP: Distributed (semi) joins (‘80s)
  - Search: MaxScore (‘90s) -> Block Max WAND (‘2010s)



Lamb, Andrew. - Dynamic Filters: Passing Information Between Operators During Execution for 25x Faster Queries

Proven: Predicate Pullup

- Pull an expensive predicate *up* the plan.
- Postgres History:
  - Starting in 1992, Postgres did “predicate migration” to pull high COST functions up.
    - Hellerstein + Stonebraker at Berkeley: supporting image processing, geo calculations
  - Removed in 1998 to reduce planner complexity.
- In Top-K evaluation, ParadeDB pulls-up visibility checks:
  - Top-K query might match millions of rows, while returning only K (e.g. `25`).
  - Overquery the LIMIT based on dead tuple fraction, defer all visibility checks until after Top-K.
  - Visibility checks rarely selective in a healthy table.
  - Reduces visibility checks, *and* the `tid` fetches themselves (late materialization!).

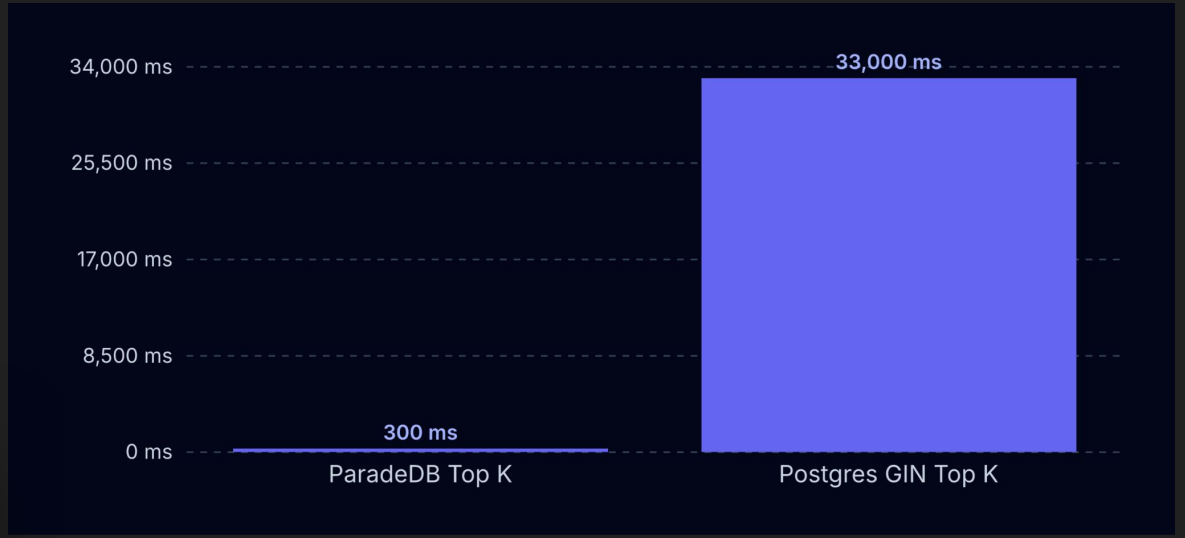
Proven: Equivalence Testing

- Heavily tested for equivalence to base scans
  - Property tested using Rust's `proptest` crate
  - Test all:
    - combinations of GUCs
    - all permutations of supported syntax, including joins
  - “Deterministically random”: failure emits a repro `regress` test
- “stressgres” tests
- Antithesis
  - All `proptest`s and stressgres tests run with Antithesis permutation of thread/lock interleavings.
- Extensive `regress` tests

Proven: Result

- For pre-filtered queries, can be dramatically faster*.
- Example:
  - Intersected prefiltering: posting list and columnar range scan
  - ORDER BY score (Block-Max WAND, dynamic filtering)
  - Late materialization
  - Predicate pullup (for visibility)

```
SELECT *, pdb.score(id) FROM benchmark_logs
WHERE severity < 3 AND message ||| 'research team'
ORDER BY pdb.score(id) DESC
LIMIT 10;
```



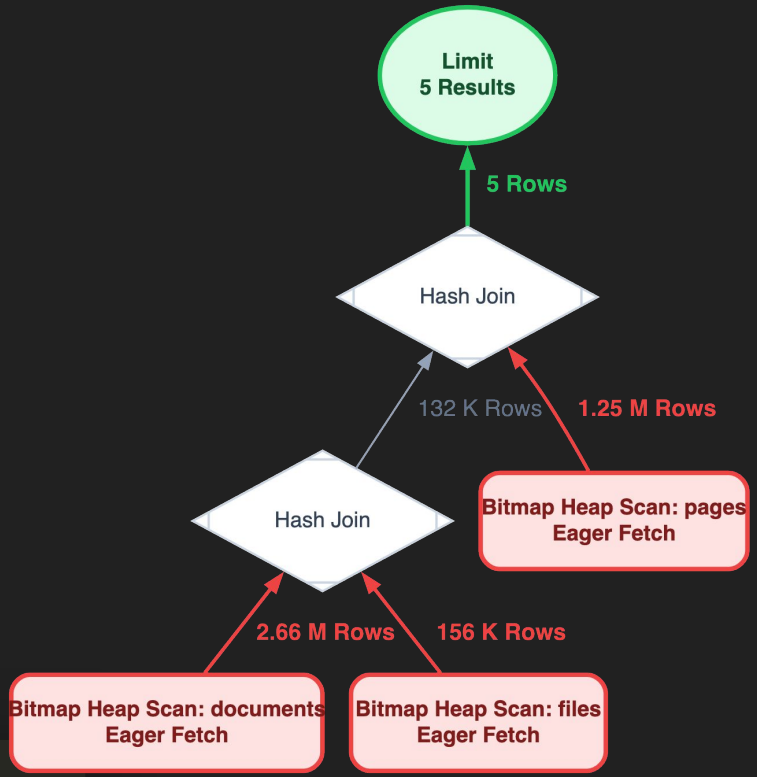
Frontier: Joins

- Fresh off the lot; Custom Scan hook:
  - `set_join_pathlist_hook` - multi-table queries
- Take over N-way joins, push down to DataFusion atop covering indexes
  - Hypothesis: apply all of these same techniques in a multi-table setting.
  - “Common knowledge” in search is that you must denormalize / use views… goal is to raise the dataset-size ceiling before that’s necessary.
- Operating entirely on Postgres planning structures (i.e. no reparsing of SQL) and buffers (i.e. no network).
  - Postgres optimizer plans the join, we execute it.
  - Upside: mature CBO!
  - Downside: must pattern-match a large variety of Postgres plan shapes.

4. Proving the Join Hypothesis

Eager Materialization Worst Case

- A worst case for eager materialization:
  - Fetch the full target list (>100 columns) for ~4mm rows before joins and Top-K.
- Without top-of-plan late materialization, DataFusion only marginally better than Postgres:
  - Can push down dynamic filters from the top, but they start out empty, and cannot be pushed through both sides of some joins.
- About 20x faster with a hand-written CTE (but code too long for slide!).
- With top-of-plan late materialization:
  - ParadeDB executes this **45x faster**, accessing 55x fewer buffers.
- **tl;dr: We really want top-of-plan late materialization.**

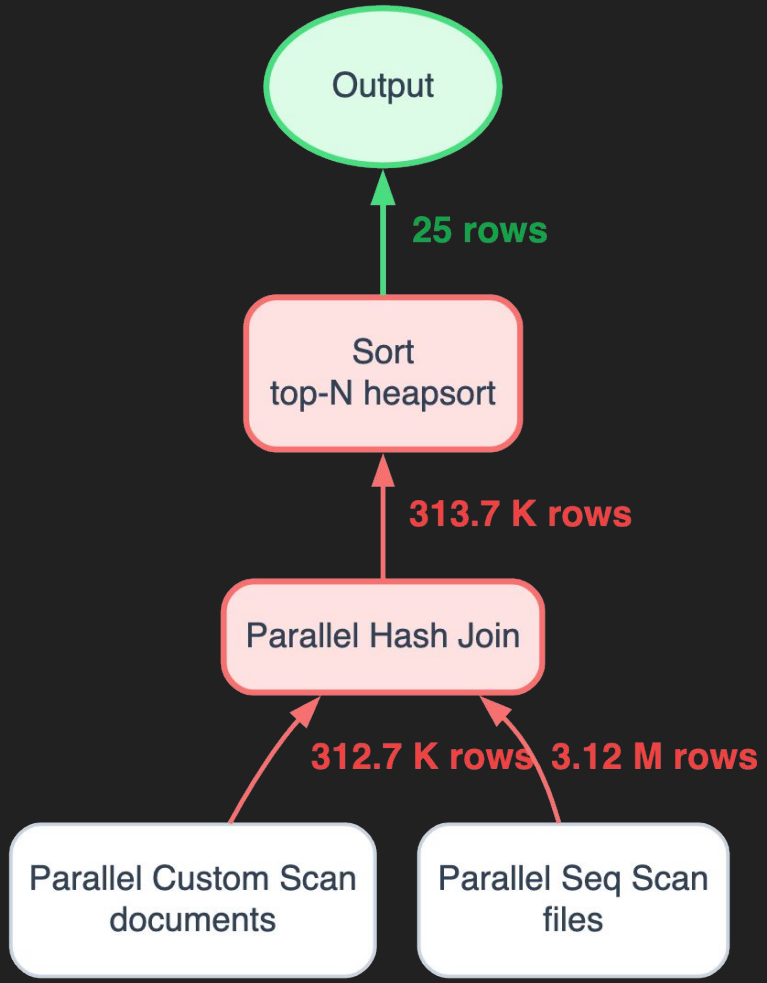



```
SELECT *
FROM documents JOIN files ON documents.id = files.documentId JOIN pages ON pages.fileId = files.id
WHERE documents.parents @@@ 'parent' AND files.title @@@ 'collab' AND pages.content @@@ 'reach'
LIMIT 5;
```

Query Walkthrough: Baseline

- Semi-join filter against a list of IDs + Top-K on a string column (note!).
- Goal: Sub-second search over a normalized relational schema.
- Baseline: 1.7s without the extension
  - Eagerly fetches all rows (445k buffers) before join, sort, limit.
  - Parallel hash join: row-wise on materialized columns.

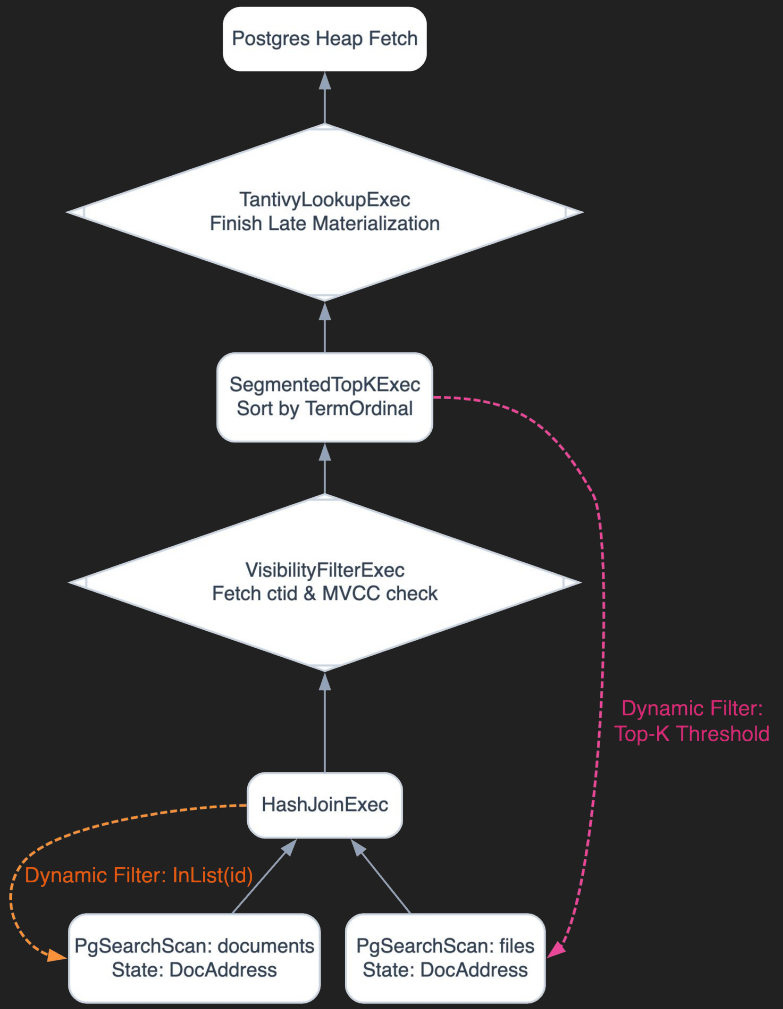
```
SELECT f.id, f.title, f.createdAt
FROM files f
WHERE f.documentId IN (
  SELECT id FROM documents WHERE parents @@@ 'alpha' AND title @@@ 'title1'
)
ORDER BY f.title ASC
LIMIT 25;
```



Query Walkthrough: ParadeDB

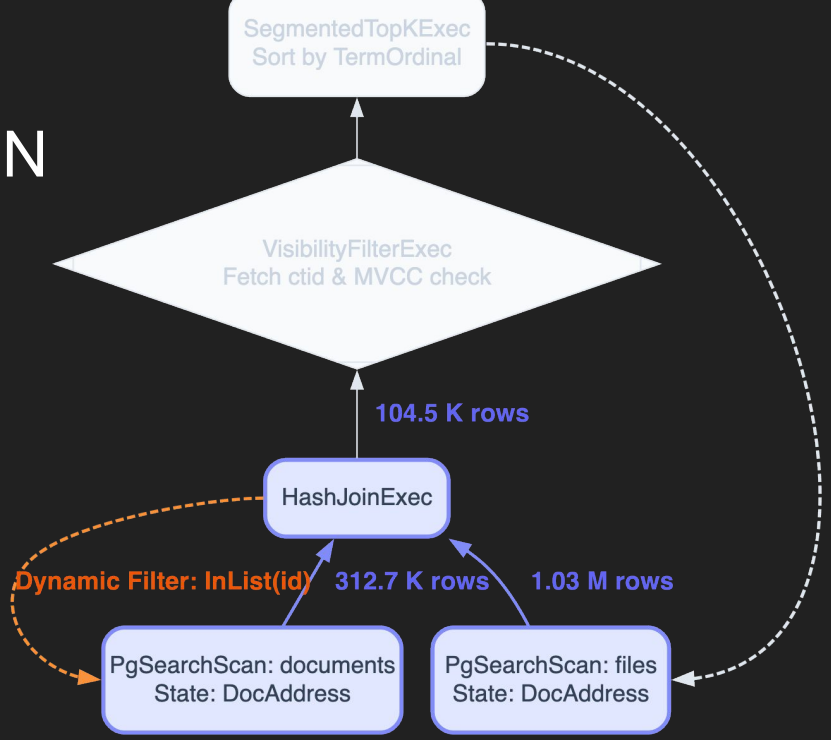
- Utilizes `set_join_pathlist_hook` to bypass the executor
  - Translate quals, target list, joins, ORDER BY, etc and then push them down
- Quals applied by Tantivy's inverted indexes as a DataFusion TableProvider
  - Implements dynamic filter pushdown for scan-internal late materialization.
- Implements a series of DataFusion logical and physical optimization passes.

```
SELECT f.id, f.title, f.createdAt
FROM files f
WHERE f.documentId IN (
  SELECT id FROM documents WHERE parents @@@ 'alpha' AND title @@@ 'title1'
)
ORDER BY f.title ASC
LIMIT 25;
```



Query Walkthrough: Scan and JOIN

- Index scans/filters on posting lists.
  - Dynamic filters from build side of HashJoin and Top-K limit.
- Late materialize / lazy decoding. Remain a union of:
  - row pointers (`DocAddress`)
  - partially materialized strings (`SegmentOrdinal + TermId`).
- Join: Column-wise `HashJoinExec`
  - Eliminates ~92% of rows.

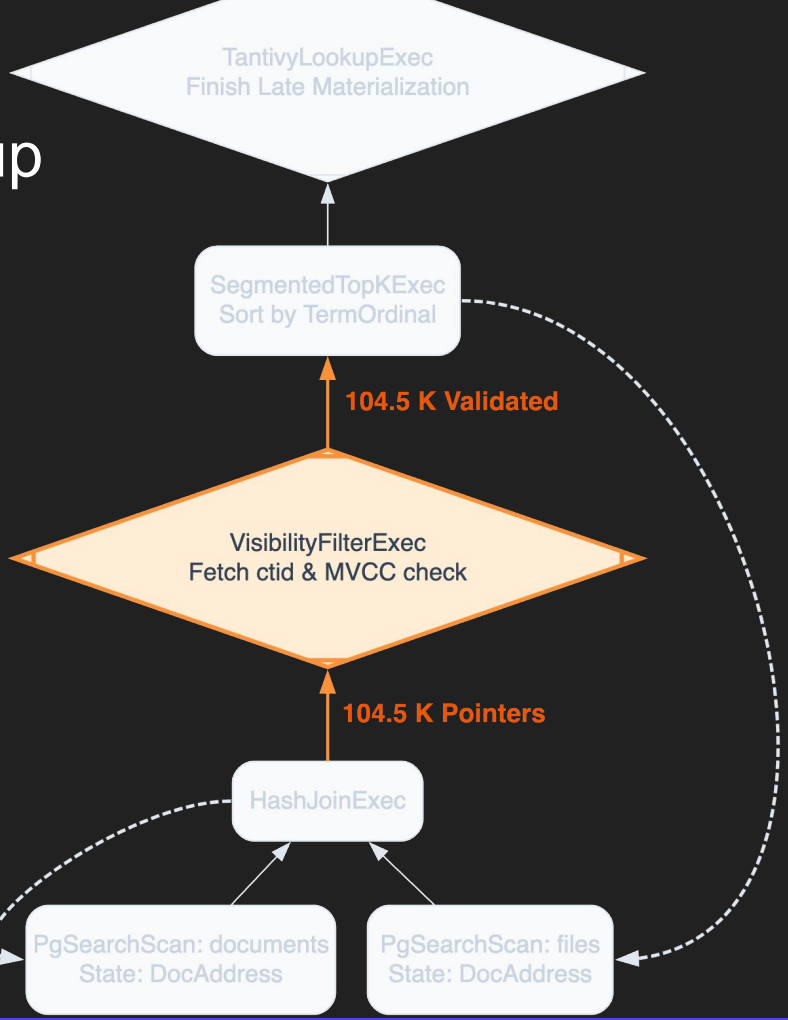



```sql
SELECT f.id, f.title, f.createdAt
FROM files f
WHERE f.documentId IN (
  SELECT id FROM documents WHERE parents @@@ 'alpha' AND title @@@ 'title1'
)
ORDER BY f.title ASC
LIMIT 25;
```

Query Walkthrough: Predicate Pullup

- Predicate pull up: safe on the preserved side(s) of the join.
  - Note: Unlike with single-table, don't pull predicate through Top-K: estimation too difficult.
  - Defers the `tid` fetch and MVCC check until *after* the join.
  - In this query: 104k rows checked rather than ~1.4m join inputs.

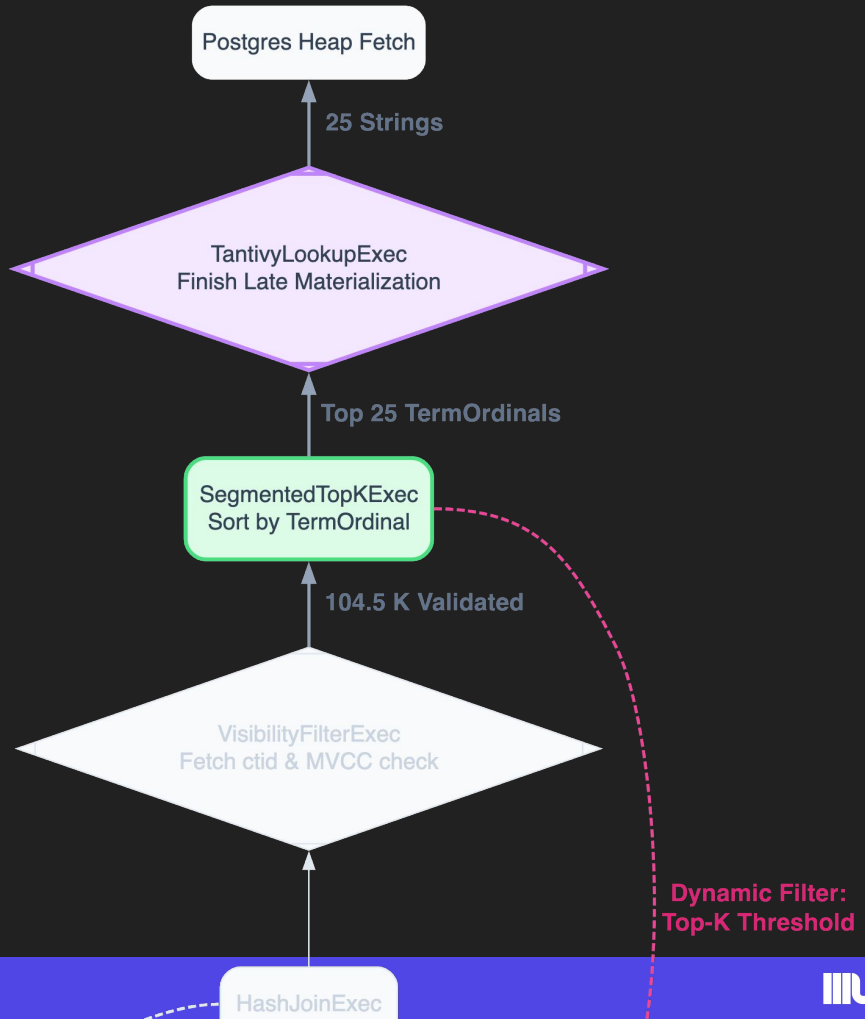
```
SELECT f.id, f.title, f.createdAt
FROM files f
WHERE f.documentId IN (
  SELECT id FROM documents WHERE parents @@@ 'alpha' AND title @@@ 'title1'
)
ORDER BY f.title ASC
LIMIT 25;
```



Query Walkthrough: Top-K

- Top-K with deferred decoding:
  - `SegmentedTopKExec`: Perform per-segment Top-K on term ids (u64), without decoding.
    - Dictionary lookups are expensive (generic compression; should probably be BtrBlocks or FSST).
  - Publish dynamic filter down for the LIMIT column.
  - Finish by decoding and emitting the global Top 25.
- Finally: `TantivyLookupExec`
  - Fetches any remaining columns.

```sql
SELECT f.id, f.title, f.createdAt
FROM files f
WHERE f.documentId IN (
  SELECT id FROM documents WHERE parents @@@ 'alpha' AND title @@@ 'title1'
)
ORDER BY f.title ASC
LIMIT 25;
```



Query Walkthrough: Result

- Optimizations:
  - Late materialization / deferred decoding
  - Dynamic filter pushdown
  - Predicate pullup
- 50318 buffers fetched (9x fewer).
- **269 ms** execution (**6.5x faster**).
  - Larger gap with a larger target list.
- Much more work to do!

```
SELECT f.id, f.title, f.createdAt
FROM files f
WHERE f.documentId IN (
  SELECT id FROM documents WHERE parents @@@ 'alpha' AND title @@@ 'title1'
)
ORDER BY f.title ASC
LIMIT 25;
```

5. API Retrospective

The Rough Edges: IAM

- Our `pdb.score` function only makes sense in the context of a specific scan.
  - No way to force projection from the IAM scan. Instead, function is an error-raising stub.
  - If it's in a target list, we *must* use a Custom Scan to implement projection.
- If IAM isn't chosen, `@@@` operator is pathological (e.g., in nested loop joins)
  - Our index is not amenable to point lookups: `Query` intersection state is heavy weight and can only be seeked forward (in index, i.e. arbitrary order).
  - Instead, executes the `Query` once and memoize the result for per-row hashmap lookup.
- We are massively multi-column, but indexes max out at 32 columns
  - Instead, must index a ROW constructor.
- All issues surmountable! But have pushed us toward Custom Scans.

The Rough Edges: Custom Scan

- Difficult / impossible to know how many parallel workers a Custom Scan gets
  - ParallelContext claims to know this, but the workers are not dedicated / may not be running your code: e.g. workers shared across both sides of a union.
  - Since cannot plan ahead, must either:
    - Dynamically consume work – usually good for latency / load balancing, but not possible for sorted scans.
    - Explicitly spawn your own (we do for aggregates) – may give a single query more resources than would be fair
- We are batched/vectorized! But `ExecCustomScan` is tuple at a time.
  - Excited about batching exposed from the IAM by the index prefetching changes.

The Good

- Custom Scan API has provided 99% of what we need.
  - ~One case where we've needed to use the planner hook: replacement of window functions.
- The Postgres extension APIs, and `pgrx` by … extension… are incredible.

Path(s) Forward?

- In flight:
  - Batching for IAMs (Peter G., Tomas V.), and for TAMs (Amit L.) … we'd love to help build it for Custom Scans.
- Thought exercises:
  - A full-fidelity alternative to Bitmap Index Scan in ctid order, with lazy visibility? Trickier without shared immutable Segments, but.
  - (Continue to) Experiment with dynamic filters?

6. Conclusion

Summary

- A Multi-Layout, Multi-Column IAM + Custom Scan allows:
  - The cheapest possible intersections of indexed predicates.
  - Deferring visibility checks.
  - Late materializing tuples.
- A TAM requires all of that, *and* requires adapting existing indexes.

Thanks! Questions?

- PGConf.dev organizers
- Open source communities
  - Postgres and pgrx
  - Tantivy (and Lucene)
  - DataFusion
- Cited researchers
- Our team at ParadeDB!
- You!
  - We're hiring.