Update on Index Prefetching

Peter Geoghegan (AWS)
Tomas Vondra (Microsoft)

PGConf.dev 2026

aws

Microsoft

Background and talk goals

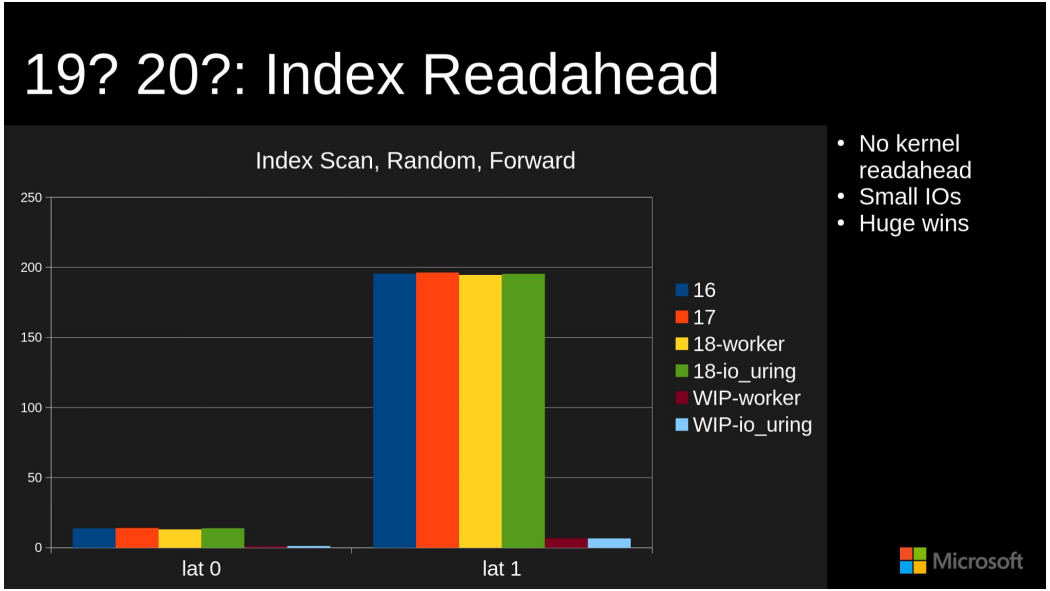
- This project was (and is) difficult and complicated
  - Several architectural issues make index prefetching during “plain” index scans (and index-only scans that require heap fetches) *much* harder than other projects that added asynchronous IO to other subsystems
  - **Yak shaving:** many of the issues we ran into were quite far removed from the core idea of prefetching using a read stream. We didn’t anticipate most of these problems.
- Patches implementing index prefetching *almost* made it into Postgres 19
- **Main focus**: the architectural challenges that led to the current design
- This talk *won’t* spend very much time on the benefits for users

aws

Microsoft

In brief: what speedups can I expect?

- Depends on workload characteristics, hardware characteristics, etc
- 10-100x speedups plausible in I/O bound cases
- [AIO in PG 18 and beyond (Andres Freund, pgconf.eu 2025)]



Agenda

- History of this project
  - First two approaches to adding index prefetching
- High-level design of latest/third approach
  - Applying the lessons from the first two failed attempts
- Regressions (and other risks)
  - How we avoided performance regressions, and managed other risks
- Future outlook
  - Follow-on work enabled by the design

History of this project

- Plain index scans = primary source of random I/O
  - Excellent target for prefetching.
- Index-only scans were always in scope
  - They *might* need to perform **heap fetches** (just like a plain index scan)
- Plain index scans are the *last* remaining scan type without prefetching
  - Sequential scans (kernel read-ahead, ~2004)
  - Bitmap Heap scans (fadvise, ~2009)
  - Postgres 18 introduced *true* AIO, which these scan types now use (via the same read stream infrastructure that the the index prefetching patchset uses)

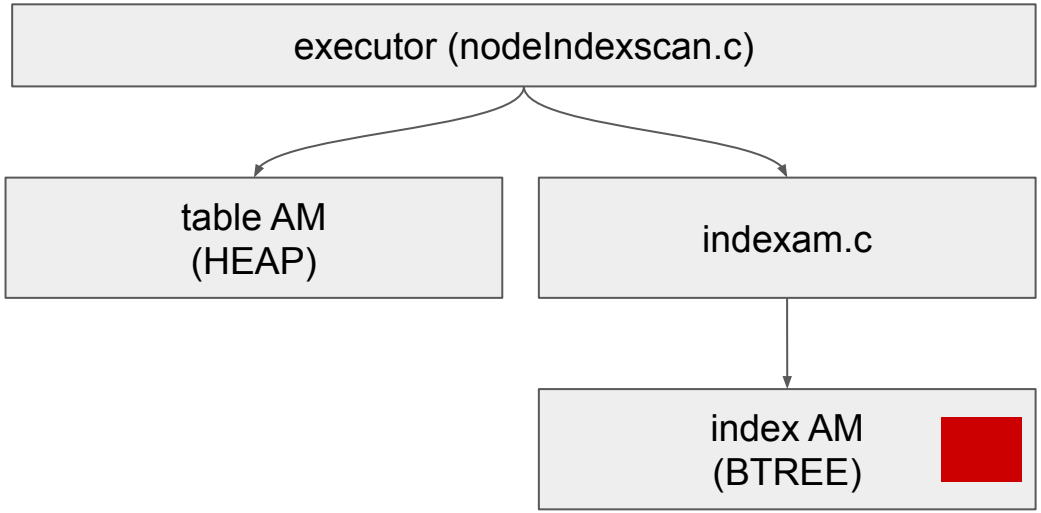
First approach: Do almost everything in the index AM

- **Minimal viable prototype** - doing posix_fadvise from the index AM
  - Simple, worked well in "good cases"
  - Predated introduction of true AIO/read stream
- First discussed at PGCon 2023 [unconference] [thread]
  - Benchmarks on the PoC patch showed promising results, good discussion starter
- Had some obvious architectural shortcomings, though
  - How far to prefetch? Requires "context" outside the AM.
  - VM checks for index-only scans? What about LIMIT queries?
- Didn’t work out

aws

Microsoft

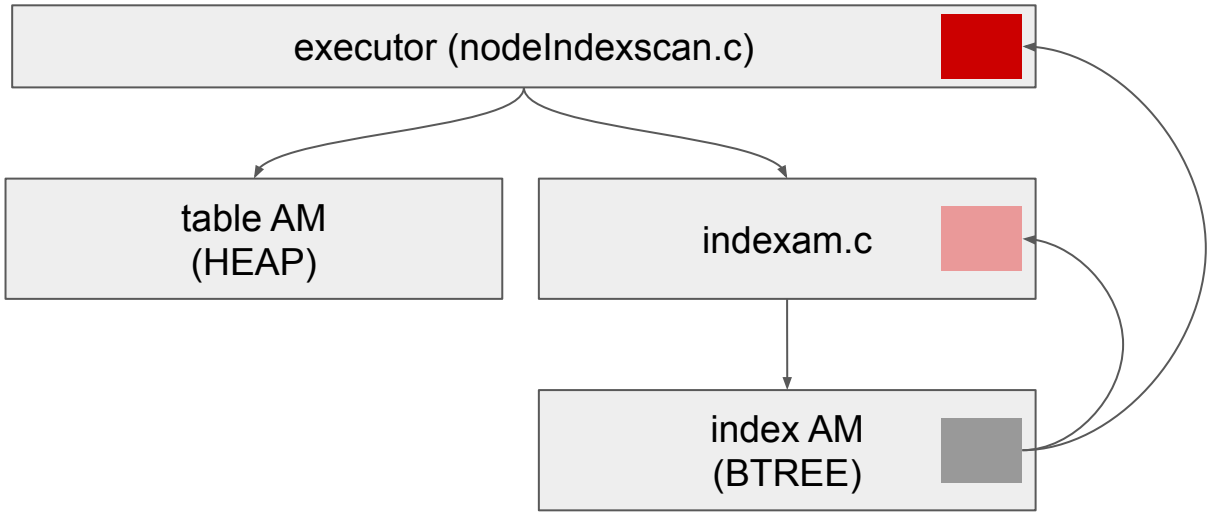
Architectural diagram (first approach)



Second approach: do "everything" in the executor

- Above AM code, doesn’t change AM code at all (uses amgettuple API)
  - Keep a queue of TIDs (and more) in the executor, feed it into a ReadStream
- **Good**: supposedly index AM agnostic (works for *any* AM with amgettuple)
- **Bad:** correctness issues (TID recycling, VM checks in IOS, kill items, ...)
- *Possibly* still workable, if we can live with certain restrictions?
  - Prefetching only within a single leaf page
  - Awkward workarounds to the correctness issues
- Didn’t work out, either

Architectural diagram (second approach)



Architectural lessons from first and second attempts

- "Minimal" redesign isn't necessarily more likely to work out
  - Attempting to keep the existing amgettuple API resulted in *awkward* coupling across layers
  - Some of the blockers were due to long standing architectural shortcomings (e.g., visibility map lookups happen in core executor's nodeIndexonlyscan.c, not in table AM)
  - Resulted in weird limitations
- Maybe we weren't ambitious enough?
  - If it *cannot* fit cleanly into the current layering scheme, a **total redesign** starts to make sense
  - How can we keep cross-layer communication/API **boundaries** relatively simple?
- "Inventor's paradox"
  - Sometimes solving a more **general** problem is ***easier*** than solving a **specific** problem

Agenda

- History of this project
  - First two approaches to adding index prefetching
- High-level design of latest/third approach
  - Applying the lessons from the first two failed attempts
- Regressions (and other risks)
  - How we avoided performance regressions, and managed other risks
- Future outlook
  - Follow-on work enabled by the design

Goals and principles behind the “third approach” design

- Important to be able to support *arbitrarily* long prefetch distances
  - Prefetching TIDs from later index leaf page tuples should be supported
  - Some workloads benefit from reading ahead by 50 or more index leaf pages
- Important that *some* layer be “in charge” – one that is **empowered** to do so
  - We **chose the table AM** – that division of labor had the fewest problems
  - Old amgettuple interface **obscures the cost** of fetching next tuple, which works against this
- Important to allow reordering of work – not *just* for prefetching
  - We must “see into the near future” to do prefetching, which also **enables other optimizations**
  - If most layers are unaware of the order that the scan performs work in, then they cannot possibly be affected by how the table AM chooses to “schedule” work – **ignorance is bliss**

aws

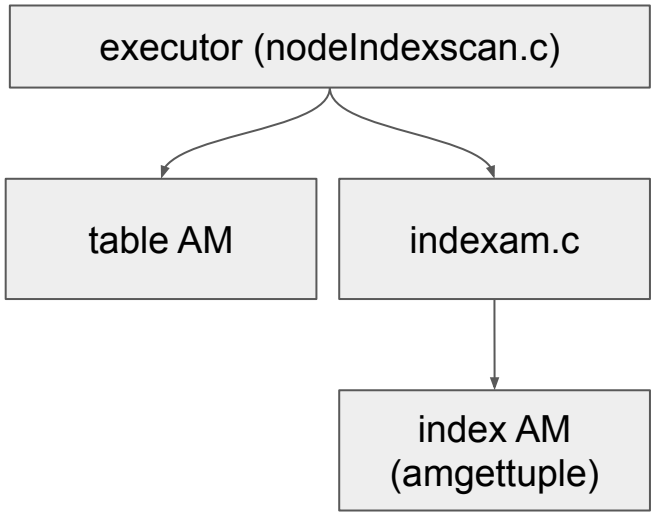
Microsoft

Division of labor under this new design (third approach)

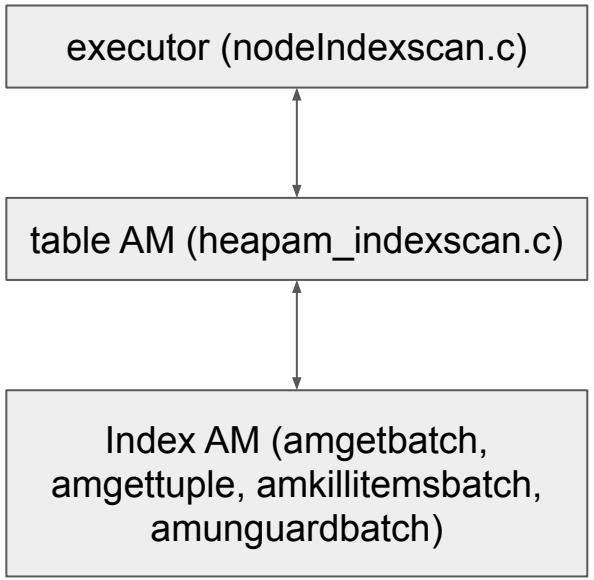
- Move things *out of* the core executor
- Move things *out of* the index AM (and restructure the API in support of this)
  - New amgetbatch interface (introduced later) exposes page layout details to table AM
- Move this functionality (from the other two layers) *into* the table AM
  - Allows the table AM (heapam) to make the best possible set of trade-offs around resource management and prefetch distance
  - As we will see, this is more complicated than you might think, once all requirements are met
- **If you're lost:** *everything* I'll say about the design from here on explains how this "division of labor" **enables** the design goals that I laid out on the last slide

High level architectural overview (third design)

Current design (Postgres 19)



Design with revised interfaces



What moves from the core executor to the table AM?

- Executor uses a higher level slot-based interface (table_index_getnext_slot), passing the scan's current direction to the table AM on each call
- All calls to the index AM are managed from *within* the table AM
- Improved layering – independently valuable (less heapam-orientated)
  - No more passing around TIDs within core executor scan nodes
  - No more visibility map lookups from nodeIndexonlyscan.c

aws

Microsoft

What is a batch? How does amgetbatch work?

- **Definition**: a batch is a group of matching index tuples/TID references that all originate from the same index leaf page
- Calling amgetbatch returns the first/next batch in line in the given scan direction (either “forwards” or “backwards”) to the table AM caller
  - Each call to amgetbatch does **the smallest possible amount of useful work** (“smallest” in terms of **CPU cycles**). Calls often just **step to the next leaf page** and return its matching items as next batch.
  - Returned batches are “owned” by the table AM. They are stored in table AM’s **ring buffer**, in index key space order.
- As we will see, amgetbatch (and related API improvements) **enable** the table AM to control certain things previously controlled by the index AM

How are index AM internals affected by batching?

- Affected index AMs now **avoid maintaining internal mutable state**, to enable **reordering work** within the table AM
- Notably, *all* **positional state** moves into the table AM
  - The table AM gets to choose whether and which “priorbatch” to pass to amgetbatch
  - Each call makes the index AM perform only the minimum amount of work required to produce the next batch of matching items
- amkillitemsbatch callback makes index AM perform LP_DEAD-marking of index tuples at a **time of the table AM’s choosing** (decoupled from scan)
  - Like amgetbatch, amkillitemsbatch provides **no high level context** about the scan – it simply instructs the index AM to LP_DEAD-mark certain index tuples (referenced from passed batch)

How are index AM internals affected by batching? (Cont.)

- Index AM *must* cede control of buffer pin resource management, too
  - Index AMs *still* required to hold on to a leaf page pin to prevent TID recycling hazards
  - But the table AM now controls *when* these interlock pins are released – **it knows best**
  - After all, the table AM (heapam) is the layer that actually needs us to avoid concurrent TID recycling hazards – it has **access** to a great deal more **relevant context**
- Index AM implements amunguardbatch callback to release buffer pin interlock
  - In nbtree and hash, amunguardbatch makes the index AM drop a pin that otherwise blocks ambulkdelete process by VACUUM (pin conflicts with cleanup locks acquired by VACUUM)
- As we will see, “ceding control” is an important part of resource management

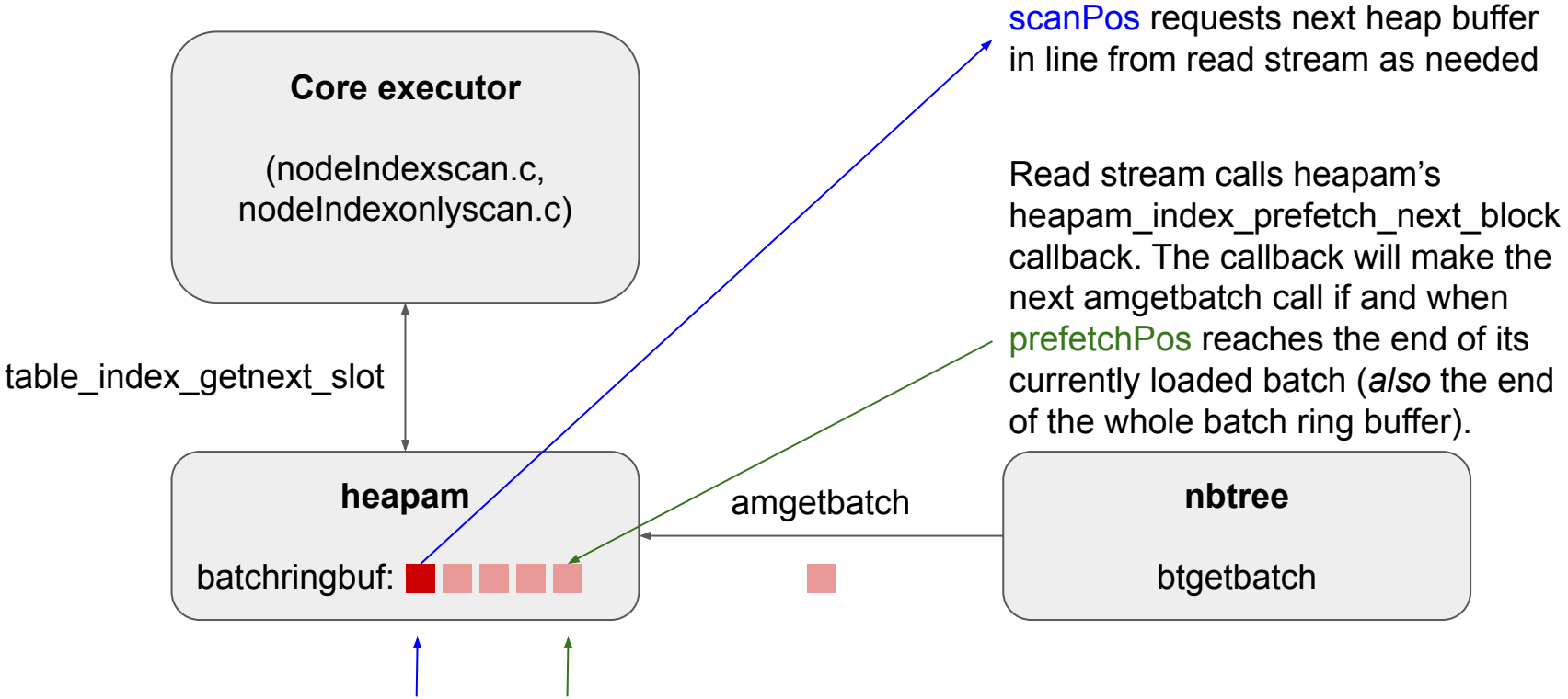
Maintaining heapam read stream when prefetching

- heapam uses **read stream** for index prefetching (just like existing aio code)
- Must be *perfectly* disciplined about “what heap blocks were requested already/will be required in the near future” to **avoid read stream disagreement**
- This is *one* reason why **all** positional state moves into the table AM
  - When scan direction changes (during a scrollable cursor scan), we need to invalidate read stream (and prefetchPos positional state along with it)
  - When prefetching begins, prefetchPos is itself initialized using scanPos positional state

aws

Microsoft

heapam's batch ring buffer and "work scheduling"



The visibility map cache's "keep read stream consistent with VM" role

- Recall that heapam (*not* the executor) now performs VM lookups during index-only scans. This is necessary to avoid scan state that **disagrees** with read stream.
  - Must determine which heap pages to **not** read at all during index-only scans **exactly once**
  - Otherwise, we have a **race condition**: the read stream *might* prefetch what turn out to be the wrong heap pages by the time the pages are consumed through scanPos.
- "Item is all-visible?" info for each item is **cached inline** (as a part of the batch itself)
- **Second role**: cached VM info is **also** used to ensure that index AMs **don't** hold on to extra buffer pins for **more than an instant,** which has certain advantages
  - Based heavily on **nbtree's dropPin optimization** (which dates back to Postgres 9.2), but *also* works during index-only scans (*all* amgetbatch scans that use an MVCC snapshot, in fact).

Problem statement: why *always* dropping index page pins eagerly was necessary (and more complicated with index-only scans that prefetch using a read stream)

- **Problem**: we'd like to drop buffer pins held on index pages right away, to avoid confusing read stream's pin management heuristics in hard-to-predict ways – ***always*** doing so **simplifies resource management**
- **Complicating factor**: Index-only scans
  - VACUUM recycles TIDs durings its third phase, which holding onto an index page pin prevents
  - Plain index scans can tolerate this, since they'll notice that any new heap tuple (at the same TID location) isn't visible to the scan's MVCC snapshot in any case – but index-only scans *need* **consistent-with-TID** visibility info
- *Theoretically* okay to not drop batch/index pins eagerly – no *formal* read stream rule forbids it
- **Conclusion**: *easier* to just invent a general solution – ***understanding*** consequences of being sloppy in some cases *harder* to make work, as a practical matter.

aws

Microsoft

The visibility map cache's **second role**: making "eager pin dropping" safe during index-only scans that need some amount of heap fetches (and that prefetch)

- **Solution enabling eager pin dropping**: index-only scans hang on to interlock pin for *just long enough* to fill the batch's VM info in batch cache
  - The amunguardbatch callback allows table AM to *slightly delay* release of interlock pin
  - In practice, read stream can *never* observe extra buffer pins held by index AM – so there's **just no question** of the **read stream heuristics** being **adversely affected by index AM pins**
- Similar TID recycling hazard when LP_DEAD-marking index tuples
  - **Solution**: index AM's amkillitemsbatch callback ***always*** performs "Has index page LSN changed?" trick (i.e. LSN check nbtree uses when dropPin=true is now used during ***all*** scans)

In conclusion: “Inversion of responsibilities” under new layering

- **Important principle**: problems became tractable once heapam was put in charge of batch resource management, and once index AM work became **“reorderable”**
- **Before**: index AM must not break heapam. Must observe **generic** interlock pin rules described by “Index Locking Considerations” docs. This was **incompatible** with prefetching TIDs from an index leaf page that is ahead of the scan position’s own leaf page.
- **After**: index AM just **does what it's told** by table AM for a given batch
- Practical lesson: sometimes you just have to **try it and see**
  - We did not anticipate that caching visibility info would simplify pin resource management
  - But we *did* have a general sense that putting the table AM in control would enable new solutions

aws

Microsoft

Agenda

- History of this project
  - First two approaches to adding index prefetching
- High-level design of latest/third approach
  - Applying the lessons from the first two failed attempts
- Regressions (and other risks)
  - How we avoided performance regressions, and managed other risks
- Future outlook
  - Follow-on work enabled by the design

aws

Microsoft

Performance regressions

- ReadStream has inherent costs
  - ReadStream initializing is not free (it’s cheap, but still has noticeable overhead with "ORDER BY … LIMIT 1" style queries)
- ReadStream prefetch distance heuristics
  - Small regressions are inevitable with *any* heuristics driven by access patterns
  - A perfect set of heuristics **does not exist** (depends on hardware, patterns ...)
  - Too aggressive vs. not aggressive enough (rarely "just right")
  - You can always can construct "adversarial" cases (dataset + query)
  - Mitigation: adjust / customize the heuristics, ...
  - While prioritizing the cases that actually matter in practice

Other risks

- New bottlenecks
  - Index prefetching generates very different I/O pattern (random + small request)
  - Pressure on different parts of the AIO infrastructure (e.g. worker queue)
- Basic correctness
  - Incorrect results obviously not acceptable
  - Mitigation: various kinds of "regression" testing
- High-level design choices
  - Maybe the design fails to consider important future requirements?
  - Biggest concern: is batch design incompatible with certain index AMs?

aws

Microsoft

Agenda

- History of this project
  - First two approaches to adding index prefetching
- High-level design of latest/third approach
  - Applying the lessons from the first two failed attempts
- Regressions (and other risks)
  - How we avoided performance regressions, and managed other risks
- Future outlook
  - Follow-on work enabled by the design

Future outlook

- Better visibility into all relevant costs at runtime
  - Allow the heuristics to make more accurate decisions (with accurate costs)
  - Make better decisions whether to use prefetching in the first place
- Improved distance heuristics in read stream
  - Custom variant for index scans? (very different access pattern)
  - Adjust distance based on I/O waits feedback? (not just after hits/misses)
- ReadStream handling of "backwards" scans
  - Allow IO combining with ascending heap blocks (just like with ascending blocks)

aws

Microsoft

Future outlook (cont. 1)

- Reducing the "unfair" preference for bitmap heap scans
  - BHS had advantage due to sequential access and prefetching
  - Maybe should adjust the costing once index scans can prefetch?
  - With index prefetching, the difference can be a lot smaller
- Improving the optimizer’s cost model
  - Costing should account for prefetching / AIO / ... (now ignored)
  - Asynchronicity = similar to parallel execution
  - Should we consider the "total" resource overhead too (not just critical path)?
- Better costing of "fun" plan differences
  - e.g. backward vs. forward scans on correlated indexes

Future outlook (cont. 2)

- Avoiding repeat locking/unlocking of the same heap page
  - Basic prototype exists already
  - Maybe even fetch heap pages *somewhat* out of order from read stream?
- Prefetching for GiST and SP-GiST scans
  - Right now only BTREE/HASH implement the amgetbatch interface
  - Not obvious how to map existing GiST concepts onto batches
    - The amgettuple property of accessing every leaf page once exists already
    - But KNN GiST ordered scans return items “out of leaf page order”
    - This complicates our approach to caching VM info during index-only scans
- Slightly easier to justify removing nodeIndexonlyscan.c with new design
  - may be a good idea for other reasons

aws

Microsoft

Thanks!