



An overview of index prefetching

Peter Geoghegan

geogpete@amazon.com
pg@bowt.ie

Overview

1. Background on Index prefetching project

Introduces basic aio/prefetching concepts

2. Bitmap scans, plain index scans, and aio

How aio works with these two similar scan types

3. Performance of index scans with asynchronous IO

Benchmark results showing the benefit of prefetching

4. Planner costing of index scans under prefetching

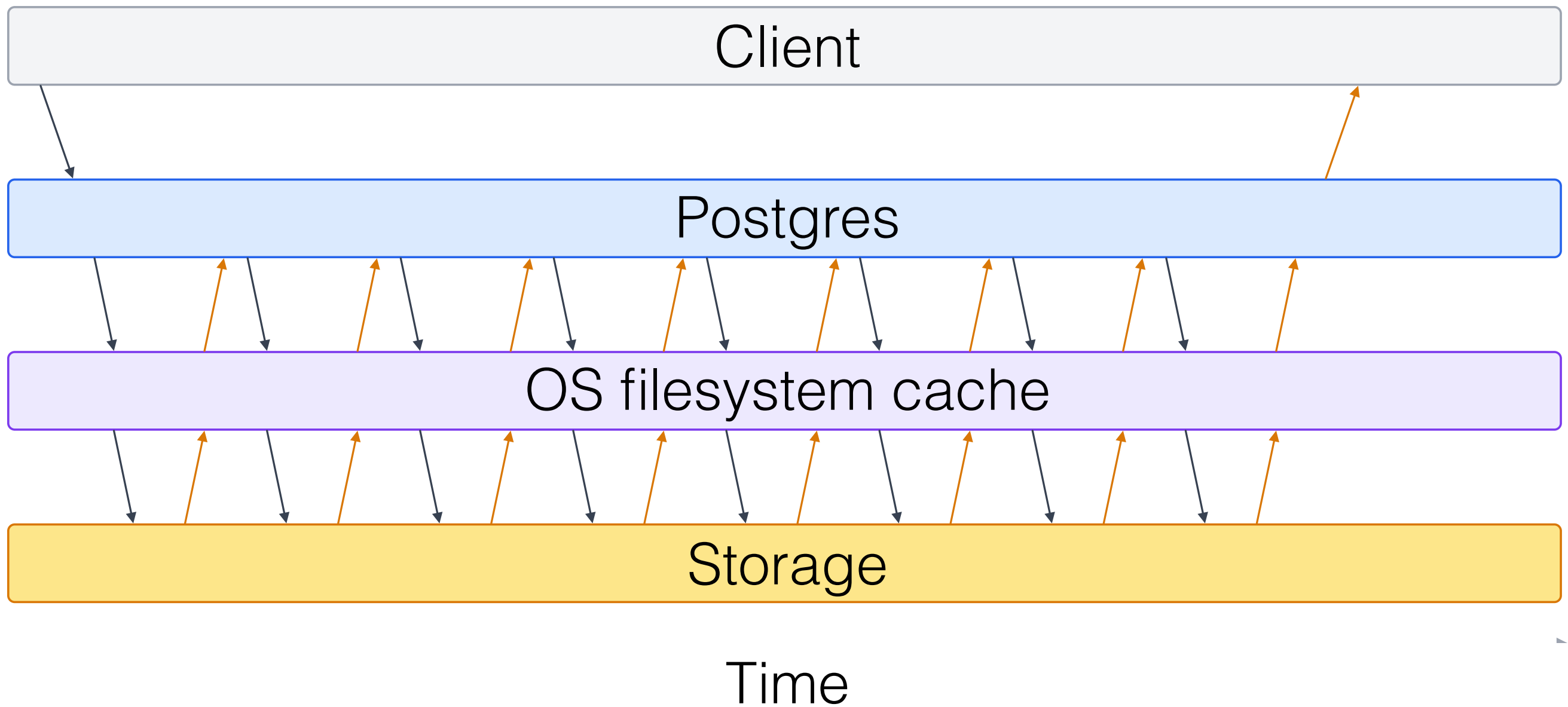
How aio affects plan costing, and the implications

Background: aio

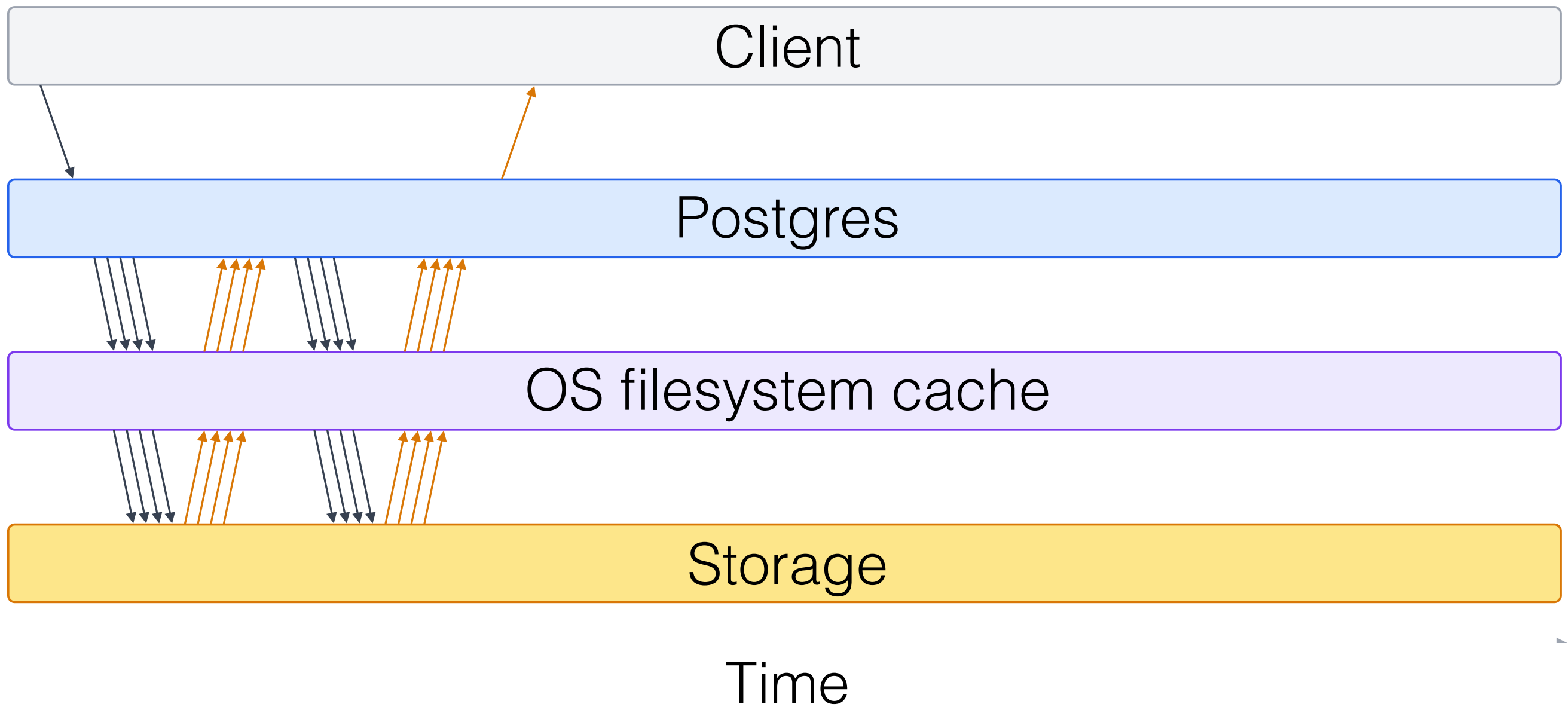
Work by Andres Freund, Thomas Munro, Melanie Plageman, and others

- Key idea: "Look into the future" to batch IO, hiding latency
 - Request the blocks that will be required ahead of time, so that by the time that they're actually needed they'll be available
 - Batches work to amortize the cost, making better use of available IO bandwidth
- Initial aio work in Postgres 18 left out support for plain index scans, which requires a fundamental rethink of the underlying design
 - However, bitmap index scans (actually, bitmap heap scans) *were* enhanced to use aio as part of that initial work





Synchronous I/O (no prefetching), uncached



Asynchronous I/O (with prefetching), uncached

Read stream aio interface

Read streams are used by every aio-aware executor node

- Registered callback requests the next heap block from table AM code
 - Returns the next required block on each call (called by the read stream)
 - *How* the callback determines which heap block will be required next varies; with index prefetching, callback might have to call *into the index AM* to figure that out
- Executor nodes call `read_stream_next_buffer()` to get a buffer containing the next page in line
 - "Next buffer" request uses same executor node's own registered callback to prefetch more blocks for later, in passing
- **Adaptive**: A miss increases the prefetch distance, avoiding future IO waits



Background: index prefetching project

Work by Tomas Vondra and myself, started by Tomas in 2023

- Builds on the asynchronous I/O infrastructure that first appeared in Postgres 18 (also uses read stream)
- Prefetches table/heap blocks (not index blocks)
- Multiple major design revisions
 - I'm optimistic that we'll commit a version with the current design before Postgres 20 feature freeze



Background: index prefetching project (cont.)

Why did index prefetching take so much longer?

- Index scan I/O requests have challenging characteristics
 - True random IO, where requests are out of order, and the same heap page can be requested multiple times
 - Simple point queries (e.g., primary key lookups) cannot benefit and must not pay a price
- Requires much closer **cooperation** between index AM and table AM
 - **Interleaved execution** (explained in the next section) required to get optimal performance
 - *All* index access methods need to adopt new batch interface: nbtree, hash, GiST, and SP-GiST all need to be switched over

Other aio/prefetching talks

- Andres Freund gave a more general and less technical talk about aio at this event last year: "AIO in Postgres 18 and beyond"
 - <https://postgresql.us/events/pgconfus2025/schedule/session/2079-aio-in-pg-18-and-beyond>
 - Available on YouTube
- Tomas Vondra and I gave a super technical talk about the underlying design at PGConf.dev 2026: "Update on Index Prefetching"
 - <https://2026.pgconf.dev/session/659>
 - Also available on YouTube

Overview

1. Background on Index prefetching project

Introduces basic aio/prefetching concepts

2. Bitmap scans, plain index scans, and aio

How aio works with these two similar scan types

3. Performance of index scans with asynchronous IO

Benchmark results showing the benefit of prefetching

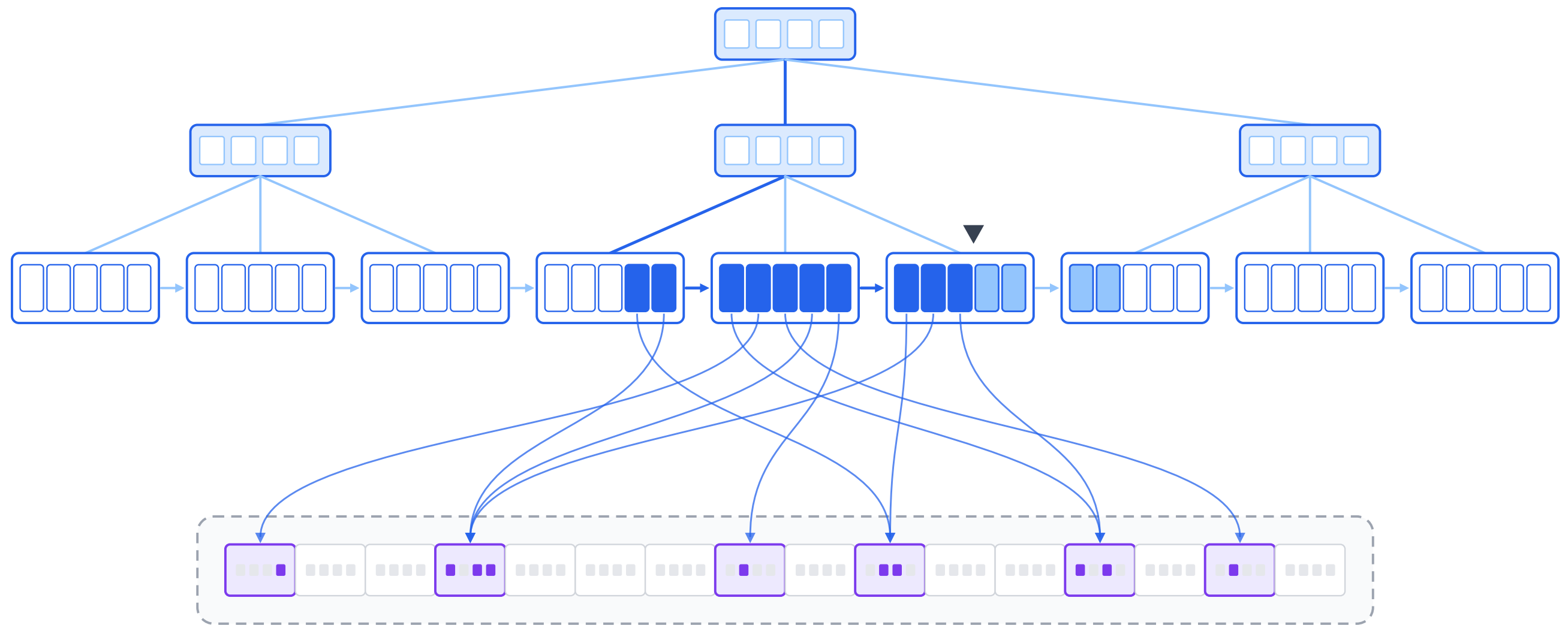
4. Planner costing of index scans under prefetching

How aio affects plan costing, and the implications

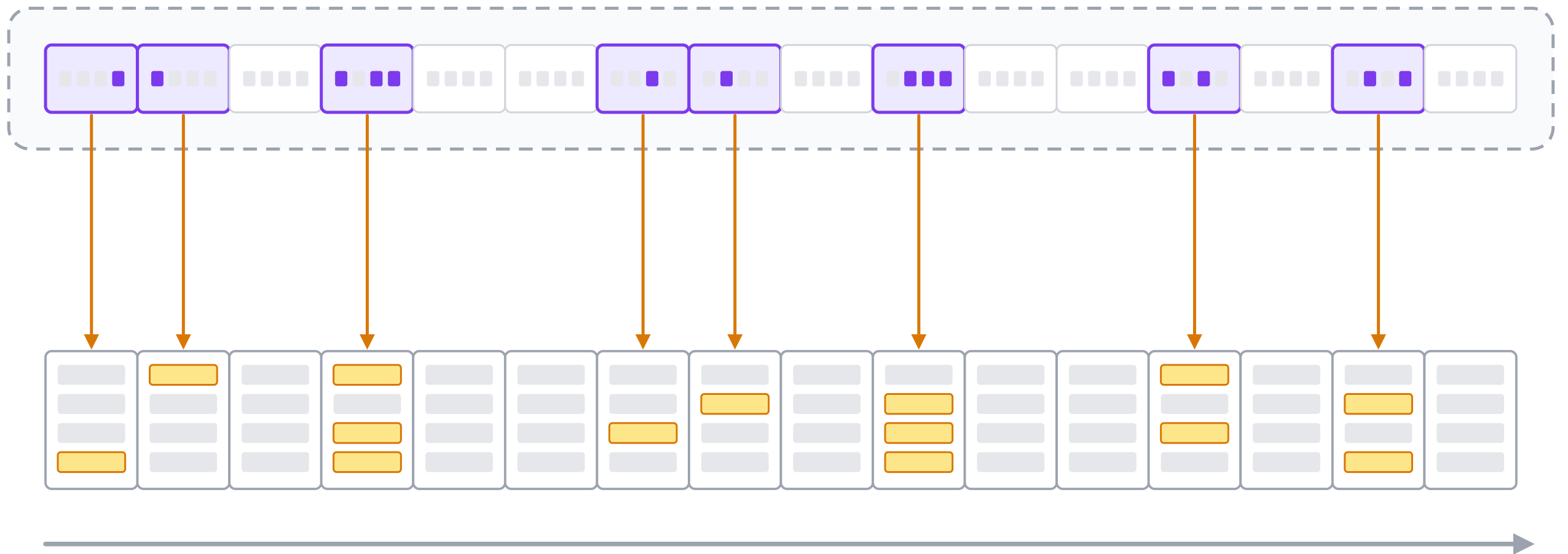
Bitmap scans

Bitmap scans always consist of (at least) two separate executor nodes, executed in **two distinct phases**

- **Bitmap index scan:** builds a bitmap representing the TIDs that the scan of the index found matched the scan condition
- **Bitmap heap scan:** uses that bitmap to actually read relevant table/heap rows
- **No overlapping** of work between these phases
 - The *earliest* point that the scan can return its first row is just after its bitmap heap scan begins
 - In other words, the bitmap index scan node has to *completely* finish *before* the first row can be returned to the client



Building our bitmap: bitmap index scan



Using our bitmap to access the heap: bitmap heap scan
(no prefetching)

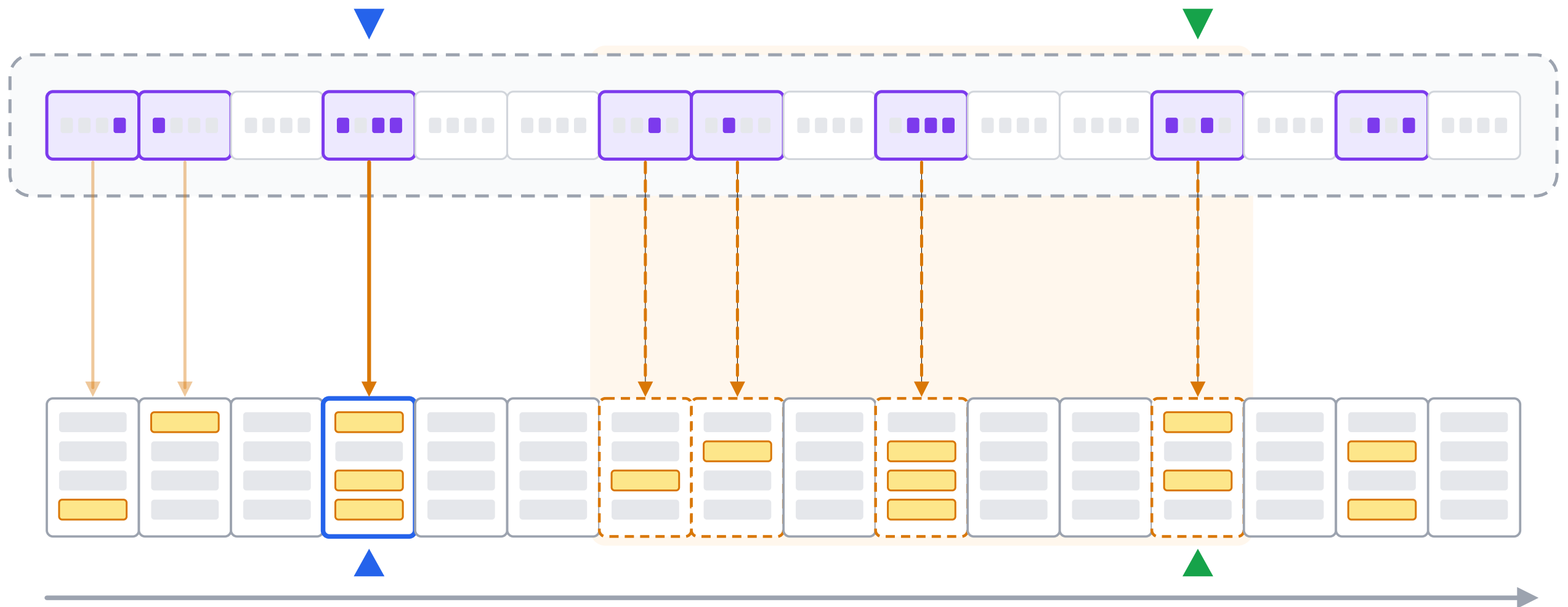
Bitmap scans: prefetching

Bitmap heap scans are ideally suited to prefetching

- Much simpler to integrate read stream than with plain index scans
 - Bitmap index scan node builds bitmap the same way regardless of prefetching
 - "Looking into the future" therefore only needs to work off of the in-memory bitmap structure; no dependencies that can get in the way
- Every required heap block is requested **exactly once**, in ascending physical heap block order; IO is "random" only to the extent that the required blocks are noncontiguous

Next buffer to
return to scan

Prefetch position

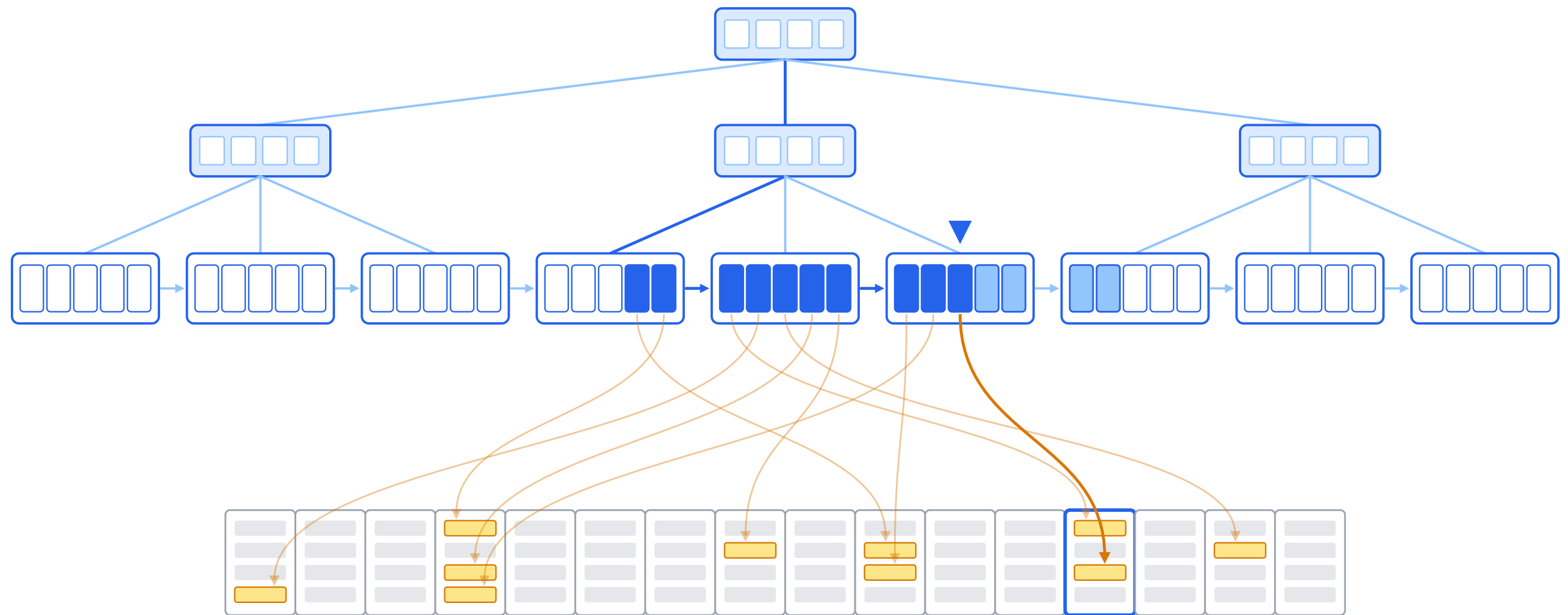


Prefetching: bitmap heap scan block requests

Plain index scans

Plain index scans access the heap/table structure incrementally, immediately after accessing the relevant index leaf page

- This is how they return results in the order that they appear in the index
 - The current position of the scan is a tuple position within an index leaf page
- Also why they require "true" random I/O to read table blocks in many cases

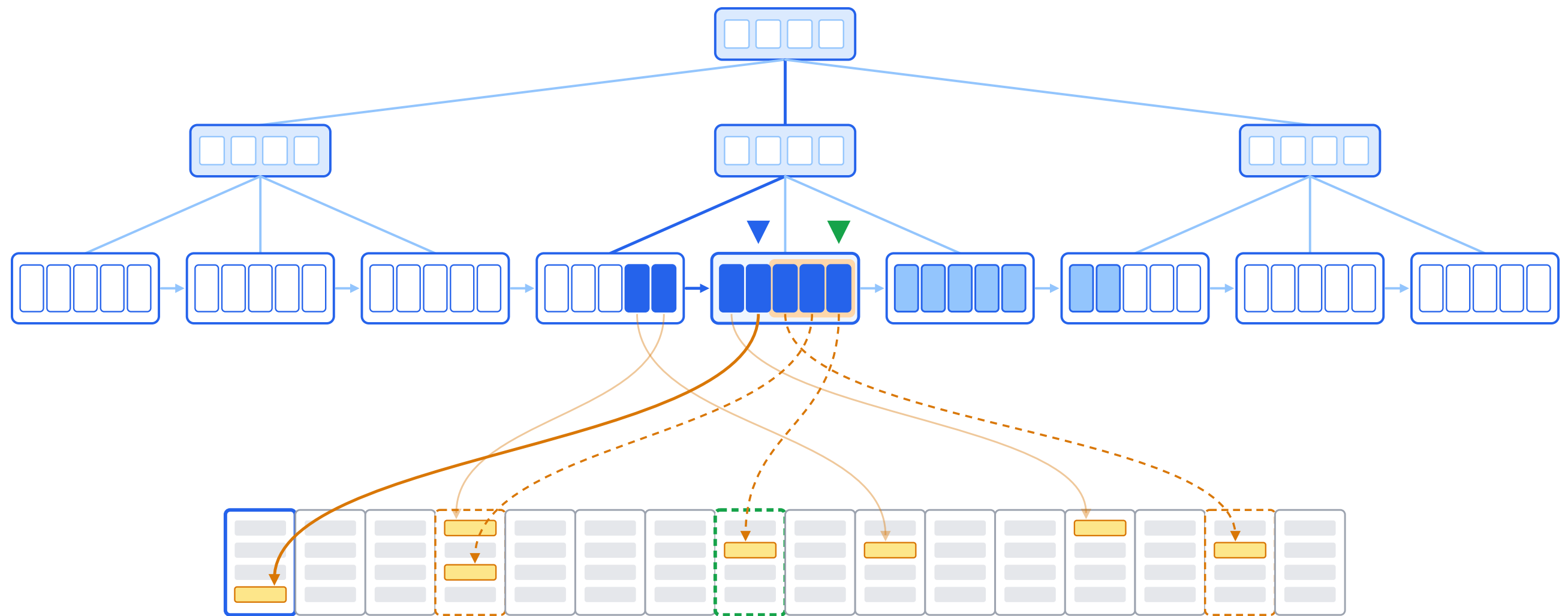


A similar plain index scan, which accesses the table/heap inline, as the index structure is scanned (no prefetching)

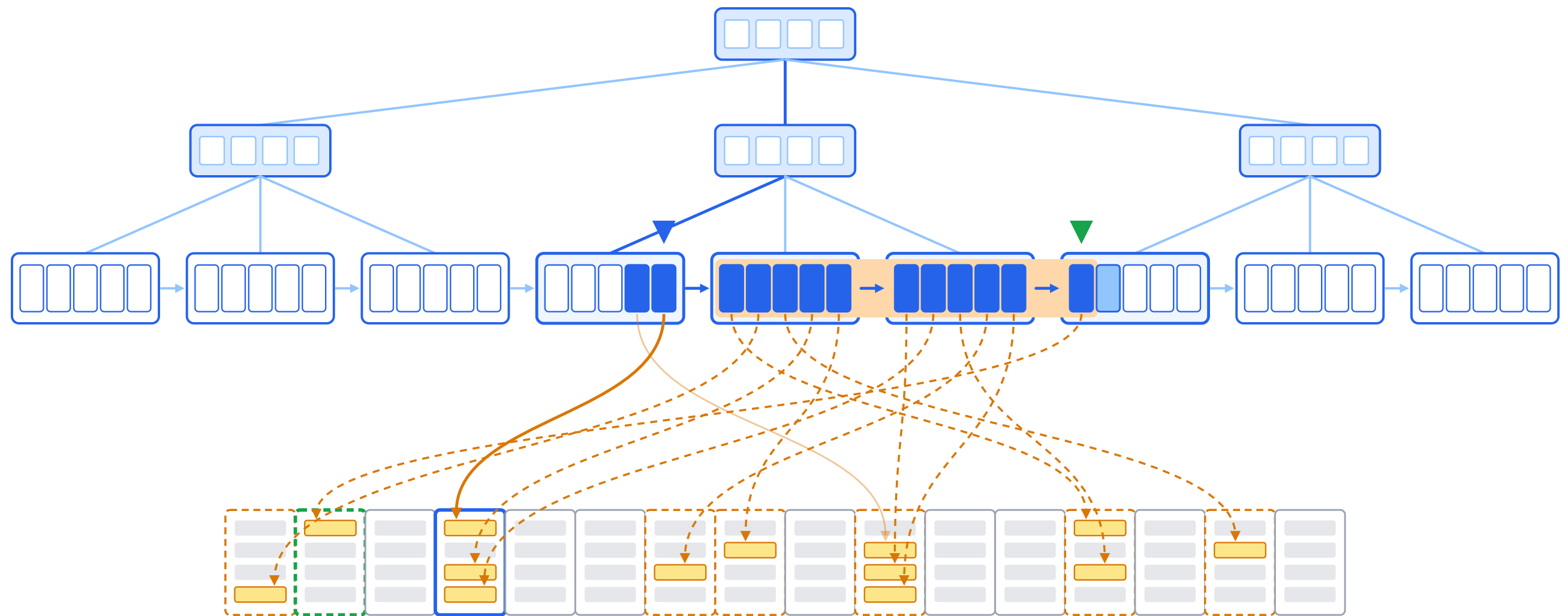
Plain index scans: prefetching

Unlike bitmap scans, prefetching during plain index scans doesn't just have a bitmap it can work off of

- **Interleaved execution:** "Looking into the future" depends on *how far ahead* we can see in the *index* structure
 - We can only prefetch what we already know will be required; figuring that out often entails reading additional *index* pages (can't just use a simple in-memory bitmap)
- Still need to return heap tuples/pages in **index scan order** (also the order that the associated TIDs are stored in the index)



Prefetching: plain index scan uses a **scan position** and a **prefetch position**, with in-flight IOs for TIDs between the two



Prefetching with **interleaved execution**: plain index scan whose **scan position** returns items on a leaf page well before **prefetch position**

Overview

1. Background on Index prefetching project

Introduces basic aio/prefetching concepts

2. Bitmap scans, plain index scans, and aio

How aio works with these two similar scan types

3. Performance of index scans with asynchronous IO

Benchmark results showing the benefit of prefetching

4. Planner costing of index scans under prefetching

How aio affects plan costing, and the implications

How much faster are plain scans with prefetching?

It depends...

- Simple point lookups won't benefit at all
 - You can't look into the future when there's only one or two heap pages involved
- More complicated queries (involving joins) benefit less when the bottleneck isn't IO waits

Benchmark details and assumptions

Index prefetching benefits uncached medium to large range scans

- There are generally 10x - 200x more heap pages involved than index pages with such queries
 - Typical of foo BETWEEN 'x' AND 'y' style range scans
 - About a million rows returned
- The numbers I'll show are for a local NVMe SSD. Network attached storage benefits even more.
- No client overhead, `effective_io_concurrency` of 100 (quite high for a production workload)

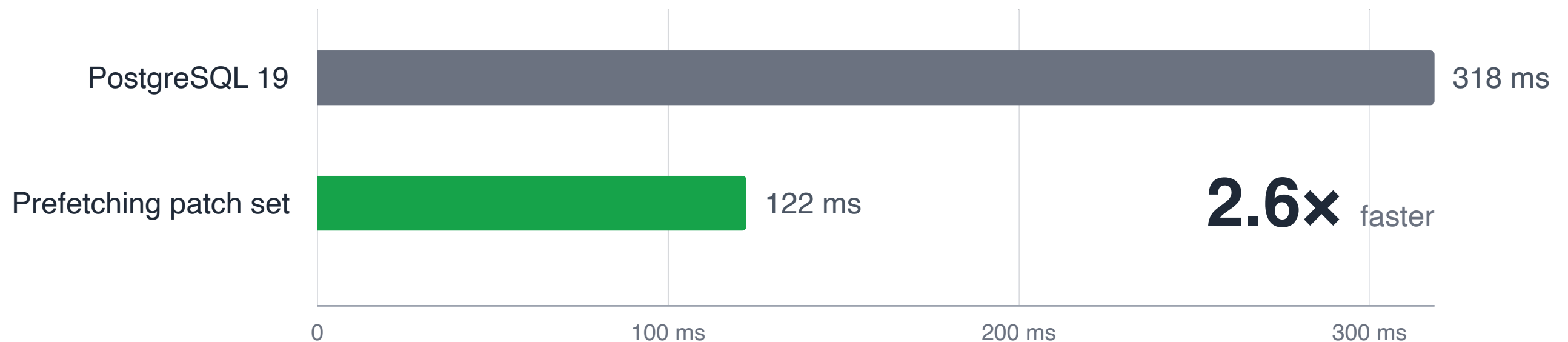


-- Sequential forwards scan

SELECT * FROM t

WHERE a BETWEEN 16336 AND 49103

ORDER BY a;

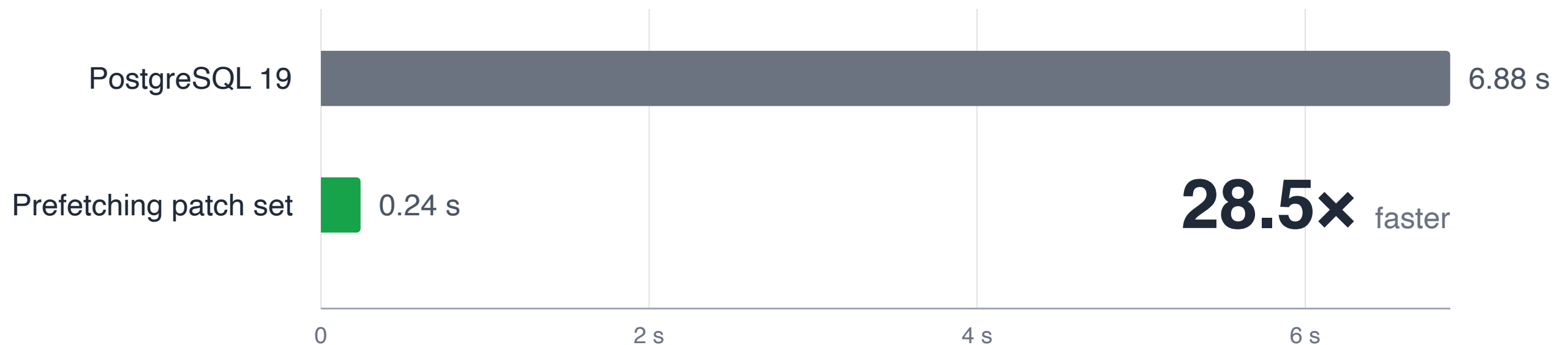


-- Sequential backwards scan

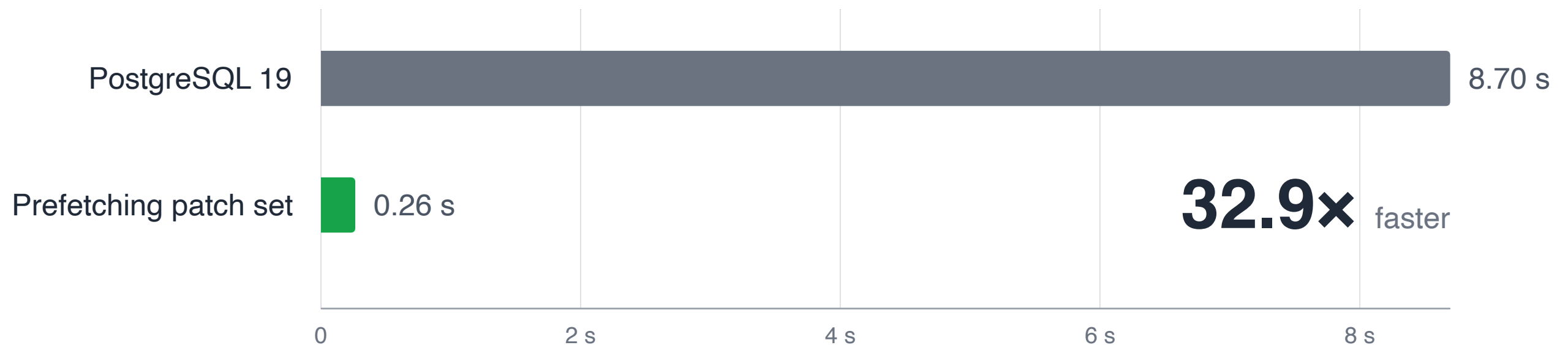
SELECT * FROM t

WHERE a BETWEEN 16336 AND 49103

ORDER BY a **DESC**;



```
-- Randomized variant scan  
SELECT * FROM t_randomized  
WHERE a BETWEEN 16336 AND 49103  
ORDER BY a;
```



Overview

1. Background on Index prefetching project

Introduces basic aio/prefetching concepts

2. Bitmap scans, plain index scans, and aio

How aio works with these two similar scan types

3. Performance of index scans with asynchronous IO

Benchmark results showing the benefit of prefetching

4. Planner costing of index scans under prefetching

How aio affects plan costing, and the implications

Planner bias in favor of bitmap scans (over plain index scans)

- Currently, the planner assumes much more of a random I/O penalty for plain index scans compared to bitmap heap scans
 - Both cost functions interpolate between `random_page_cost` and `seq_page_cost`
 - But only `cost_bitmap_heap_scan` does so in a way that models IO costs as more sequential (i.e. lower) as the scan reads more of the table
- Prefetching **justifies** costing plain index scans in roughly the same way, eliminating the bias
 - Setting `random_page_cost` to 1.0 is a **risky** way to get roughly the same behavior today. Improves performance with **fully cached** data, where plain index scans tend to be faster (much *slower* otherwise).

Planner costing weaknesses around cache hit rate

- The planner has only a **rudimentary understanding** of caching by `shared_buffers`
 - Mostly just applies `random_page_cost` and `seq_page_cost`, without really attempting to model caching at execution time for a specific query's access path
- Clearly uncached plain scans are much slower without prefetching, so the historical bias makes *some* sense - though it's **often wrong** in practice!

Traditional advantages of plain index scans over bitmap scans

- Plain scans return their results in B-Tree/sorted order, **avoiding** separate **sort node**
 - Much lower startup costs/time to return the first result row. `ORDER BY foo_column LIMIT 10` style queries particularly likely to benefit from this.
 - Can feed into a parent `GroupAggregate` or `MergeJoin` node
- Often, when `random_page_cost` of 1.0 helps, it is due to these factors (plus a fully cached working set). We'll see an example of this shortly.

-- Plain scan versus bitmap scan test query:

```
EXPLAIN (ANALYZE, BUFFERS, IO)
  SELECT * FROM t_randomized
  WHERE a BETWEEN 16336 AND 49103
  ORDER BY a;
```

- This ORDER BY range scan query will be used to compare plain scans to bitmap scans, with and without fully cached data
 - **Same query and setup** as earlier benchmark query
 - Random I/O, with limited opportunities to combine adjacent block requests
 - In practice, Postgres 19 planner always picks bitmap scan
- Simplicity is an advantage with this microbenchmark
 - Less noise, more signal. Directionally correct is good enough.

-- Plain scan on Postgres 19, fully cached:

Index Scan using t_randomized_idx on t_randomized (cost=...) (actual...
Index Cond: ((a >= 16336) AND (a <= 49103))
Index Searches: 1
Buffers: shared hit=83898
Execution Time: **124.725 ms**

-- Bitmap scan on Postgres 19, fully cached:

Sort (cost=...) (actual time=...)
Sort Key: a
Sort Method: quicksort Memory: 114689kB
Buffers: shared hit=80687
-> Bitmap Heap Scan on t_randomized (cost=...) (actual time=...)
Recheck Cond: ((a >= 16336) AND (a <= 49103))
Heap Blocks: exact=77813
Prefetch: avg=1.00 max=1 capacity=1616
Buffers: shared hit=80684
-> Bitmap Index Scan on t_randomized_idx (cost=...
Index Cond: ((a >= 16336) AND (a <= 49103))
Index Searches: 1
Buffers: shared hit=2871
Execution Time: **256.244 ms**



Traditional advantages of bitmap scans over plain index scans

- Most notably, the ability to prefetch
 - The next slide will show just how **huge** this traditional advantage has been; planner's "risk aversion" was well founded
 - This tended to drown out the traditional advantages of plain scans during planning, even **with** cached data
- Only bitmap scans can easily be **combined** using a separate `BitmapAnd` or `BitmapOr` node (not relevant to these test queries)

-- Plain scan on Postgres 19, table completely uncached:

Index Scan using t_randomized_idx on t_randomized (cost=...) (actual time=...)
Index Cond: ((a >= 16336) AND (a <= 49103))
Index Searches: 1
Buffers: shared hit=6085 read=77813
Execution Time: **8590.408 ms**

-- Bitmap scan on Postgres 19, table completely uncached:

Sort (cost=...) (actual time=...)
Sort Key: a
Sort Method: quicksort Memory: 114689kB
Buffers: shared hit=2874 read=77813
-> Bitmap Heap Scan on t_randomized (cost=...) (actual time=...)
Recheck Cond: ((a >= 16336) AND (a <= 49103))
Heap Blocks: exact=77813
Prefetch: avg=269.39 max=317 capacity=1616
I/O: count=28123 waits=11 size=2.77 in-progress=97.16
Buffers: shared hit=2871 read=77813
-> Bitmap Index Scan on t_randomized_idx (cost=...) (actual time=...)
Index Cond: ((a >= 16336) AND (a <= 49103))
Index Searches: 1
Buffers: shared hit=2871
Execution Time: **377.832 ms**



-- Plain scan with index prefetching, table completely uncached:

Index Scan using t_randomized_idx on t_randomized (cost=...) (actual time=...)
Index Cond: ((a >= 16336) AND (a <= 49103))
Index Searches: 1
Prefetch: avg=247.70 max=268 capacity=1616
I/O: count=32720 waits=554 size=2.38 in-progress=99.74
Buffers: shared hit=6085 read=77813
Execution Time: **274.795 ms**

-- Bitmap scan on Postgres 19, table completely uncached (unchanged by patch):

Sort (cost=...) (actual time=...)
Sort Key: a
Sort Method: quicksort Memory: 114689kB
Buffers: shared hit=2874 read=77813
-> Bitmap Heap Scan on t_randomized (cost=...) (actual time=...)
Recheck Cond: ((a >= 16336) AND (a <= 49103))
Heap Blocks: exact=77813
Prefetch: avg=269.39 max=317 capacity=1616
I/O: count=28123 waits=11 size=2.77 in-progress=97.16
Buffers: shared hit=2871 read=77813
-> Bitmap Index Scan on t_randomized_idx (cost=...) (actual time=...)
Index Cond: ((a >= 16336) AND (a <= 49103))
Index Searches: 1
Buffers: shared hit=2871
Execution Time: **377.832 ms**



Getting the best of both worlds with index prefetching

	Fully cached	Heap uncached	Uncached slowdown
Bitmap heap scan + Sort, both versions	256 ms	378 ms	~1.5x
Plain index scan, PG 19	125 ms	8590 ms	~69x
Plain index scan, master + prefetching	125 ms	275 ms	~2.2x

Getting the best of both worlds with index prefetching (cont.)

- Prefetching driven by a bitmap is *bound* to always have certain advantages over prefetching that returns results in some other order
 - Especially when this "other order" happens to be a totally **random** one
 - From a storage device utilization perspective, bitmap scans *still* have real advantages
 - But the added cost of random I/O is much less of a problem for plain scans when they can **hide the added latency** in the background
- Plain scans *still* have traditional advantages (e.g., avoiding sort node)
- **Conclusion:** With prefetching, plain index scans can *at least* be much more competitive, even when increased random IO is required. This is likely to enable better plans for many real world applications.

What prefetching lets the planner do

- Prefetching will enable costing resulting in plain index scans being chosen (particularly over bitmap scans) more often
 - Lowering the planner's cost estimate for index scans will *indirectly* improve performance, regardless of caching
 - Proposal on how exactly this will work is under discussion
- Improved cost profile (i.e. "uncached slowdown" ratio) makes life easier for the planner
 - Compensates for the planner having only a rudimentary understanding of `shared_buffers` hit rate

Why not improve how planner models caching by making it *more accurate*?

- It's certainly possible, but fraught with difficulties...
- Statistics can't be expected to work very well for this.
 - How can we **inexpensively** represent the current contents of `shared_buffers`?
 - Unlike `pg_stats` style statistics (which are used by the planner to generate cardinality estimates), the contents of `shared_buffers` changes **quickly** and **unpredictably**
- But improving the cost model is much easier once query execution has a simpler, more **forgiving** cost profile

Conclusions

The basic design of plain index scans had to change to make prefetching work well

Performance improvements of 30x or more are quite possible

For reasons that are fundamental to how the various scan nodes work, plain index scans get outsized benefits

This will have nonobvious impact on how the planner assesses the cost of competing scan nodes, even with applications/databases with fully cached working sets

