



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

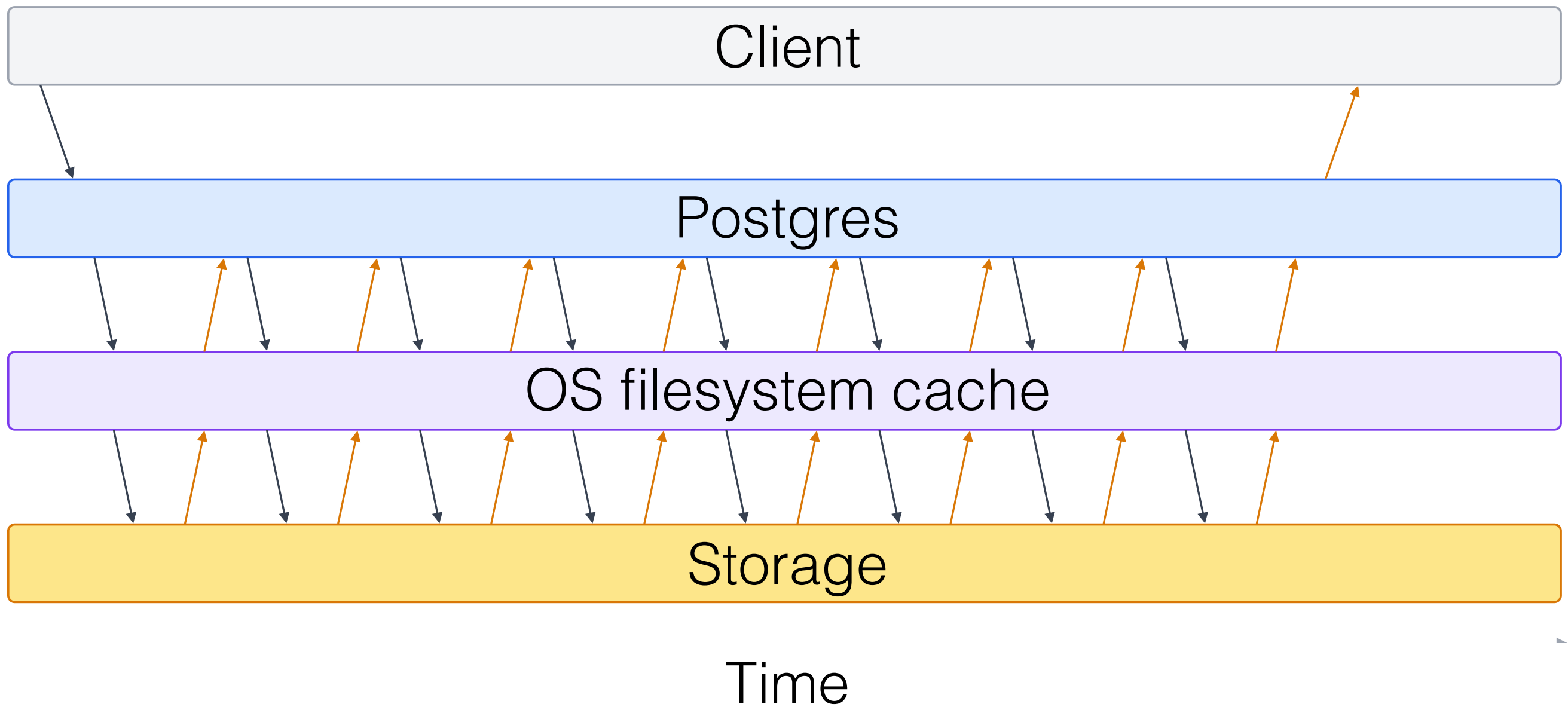
Benchmark results showing the benefit of prefetching

Background: aio

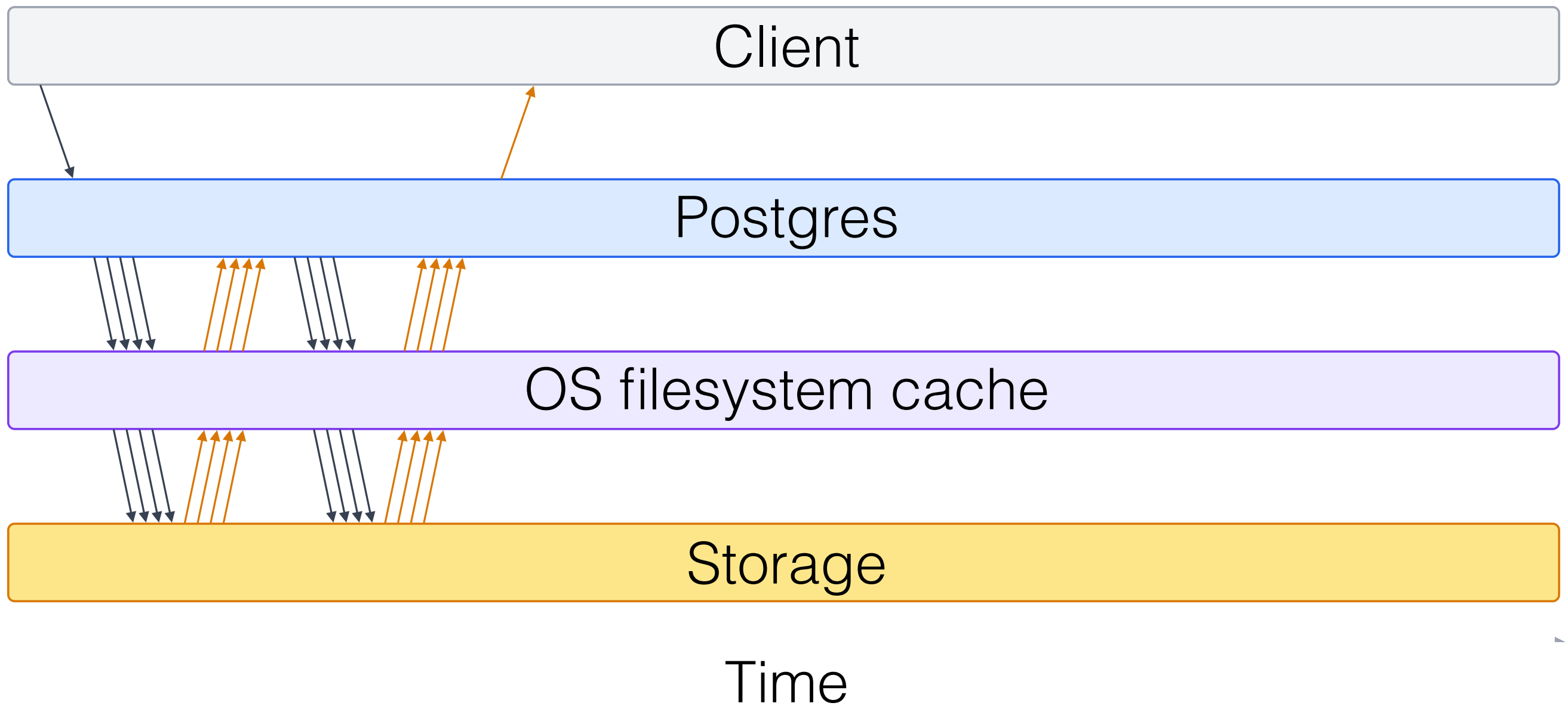
Work by Andres Freund, Thomas Munro, Melanie Plageman, and others

- 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

Background: index prefetching project

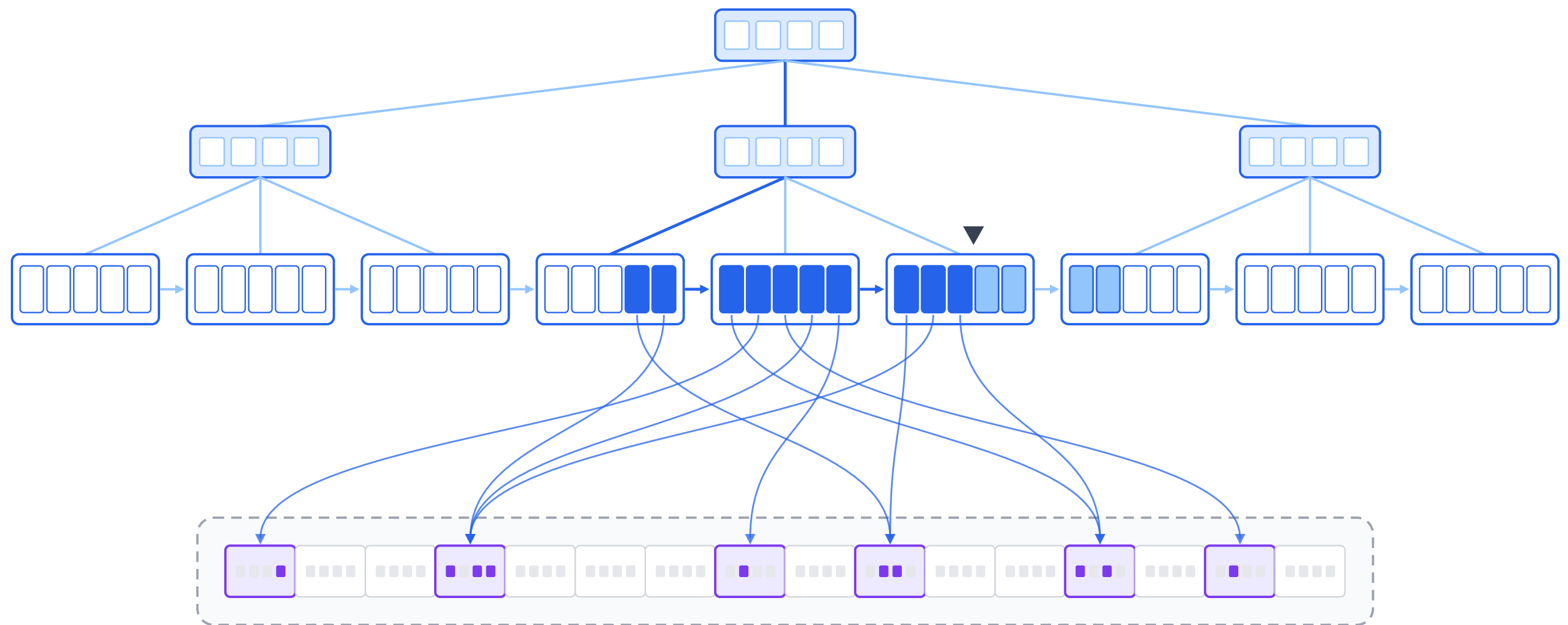
Work by Tomas Vondra and myself, started by Tomas in 2023

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

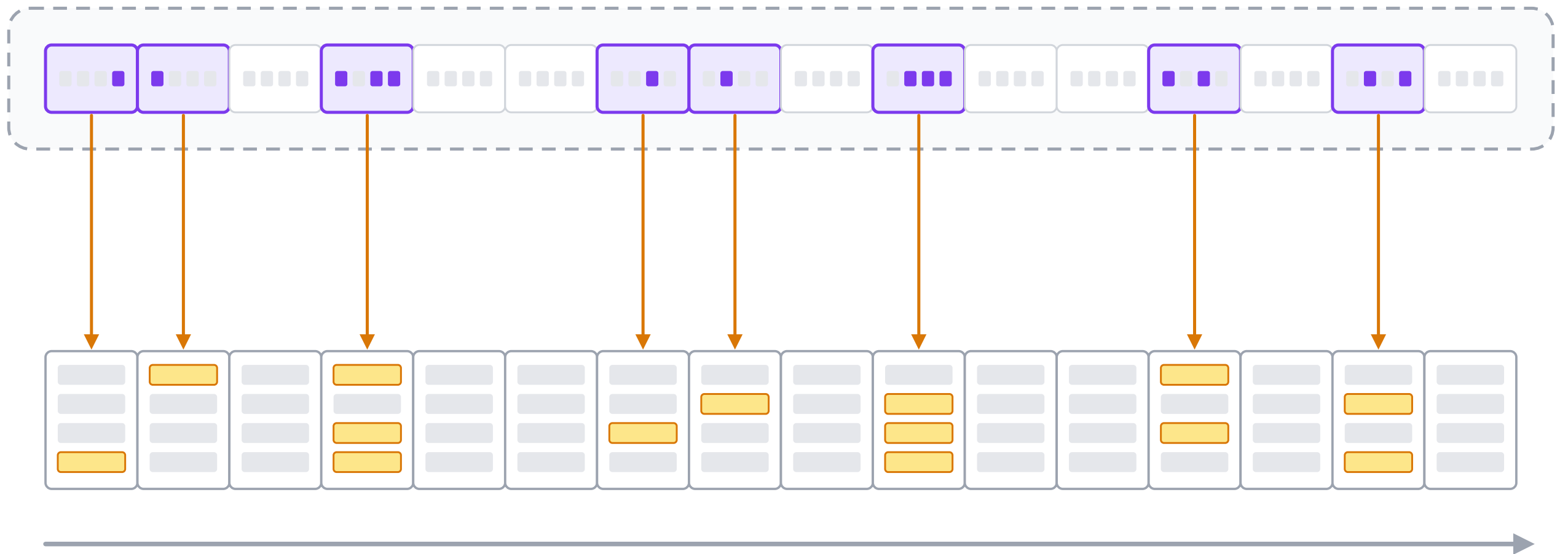
Bitmap scans

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)

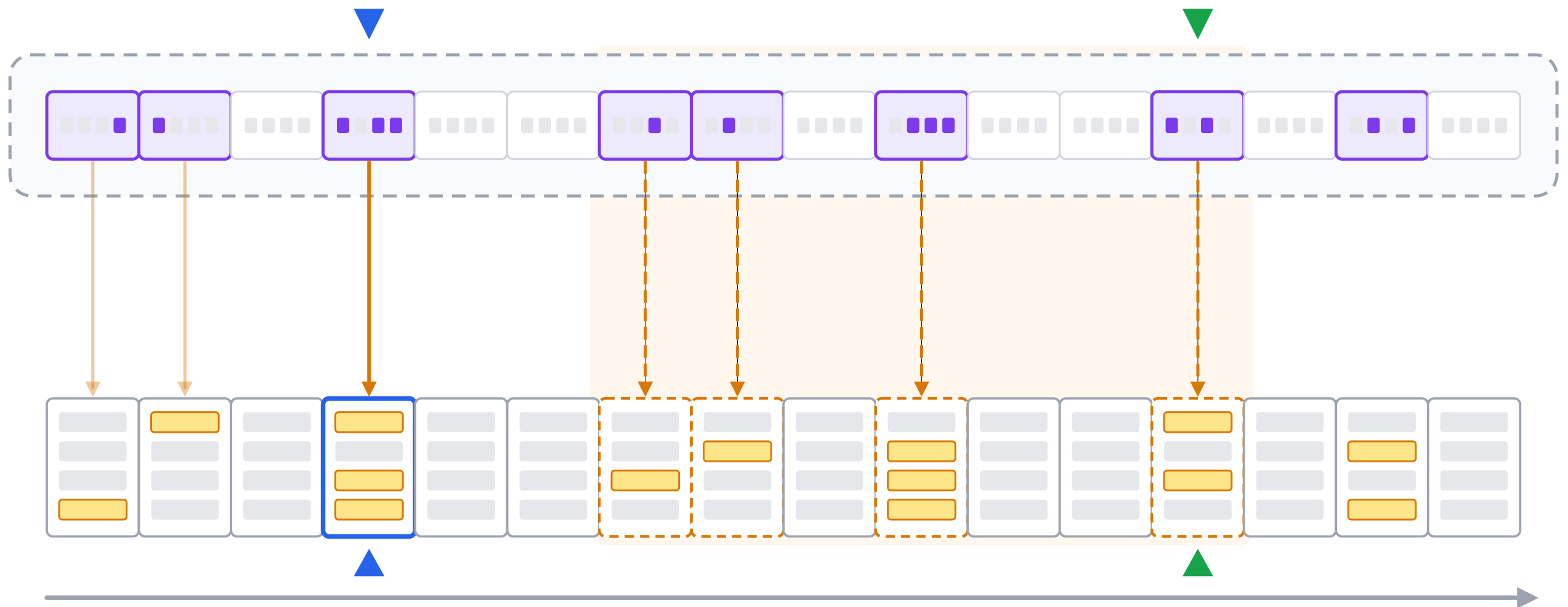
Bitmap scans: prefetching

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

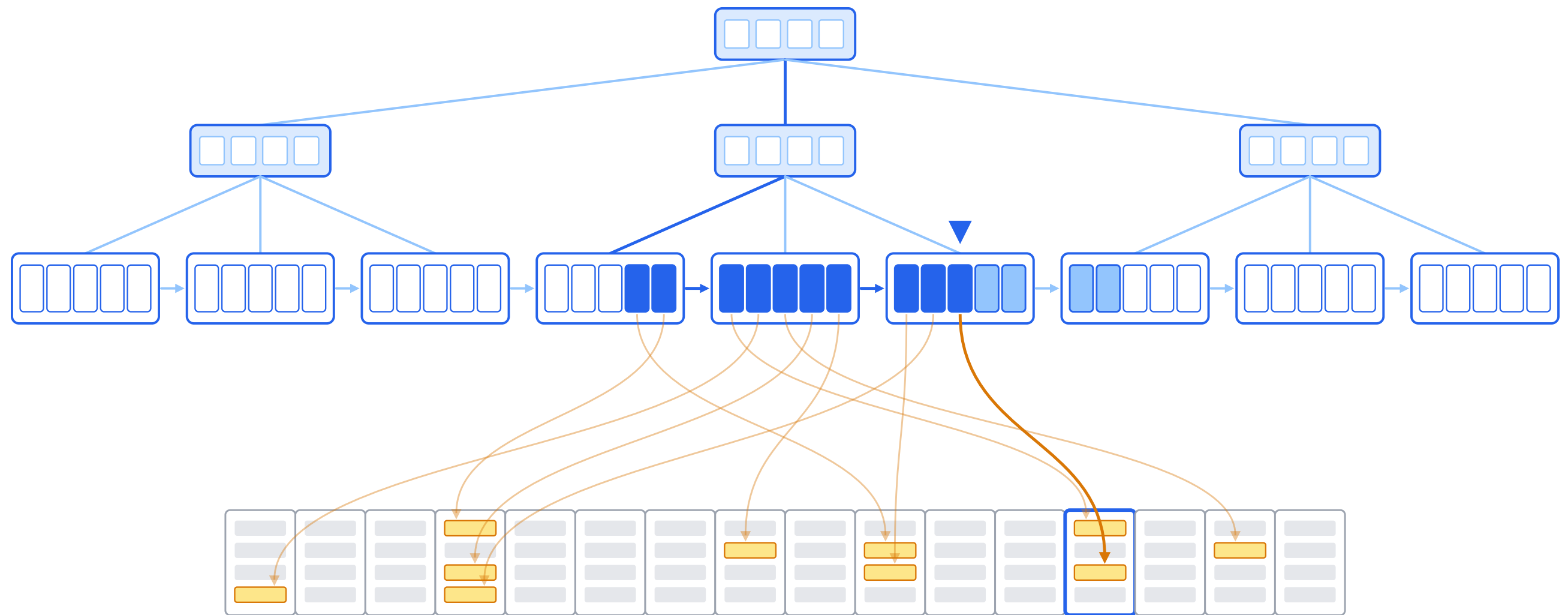


Prefetching: bitmap heap scan block requests

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

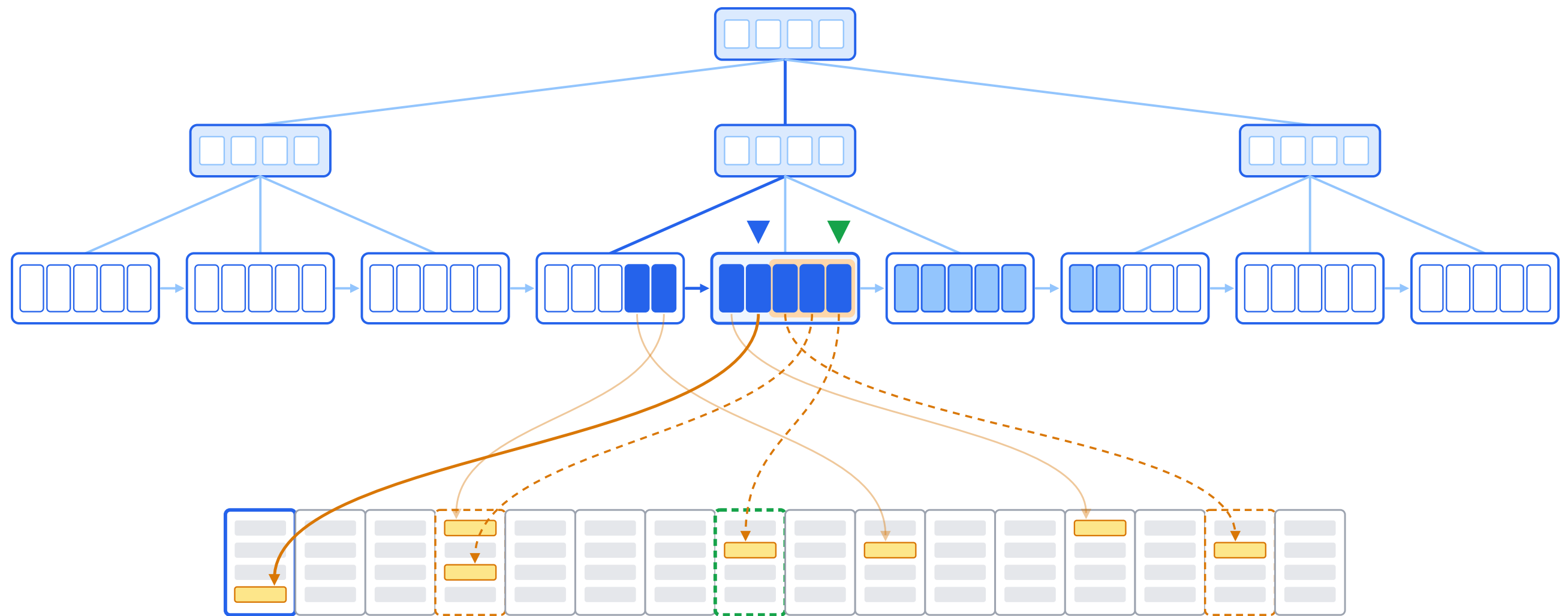


A similar plain index scan, which accesses the table/heap inline, as the index structure is scanned (no prefetching)

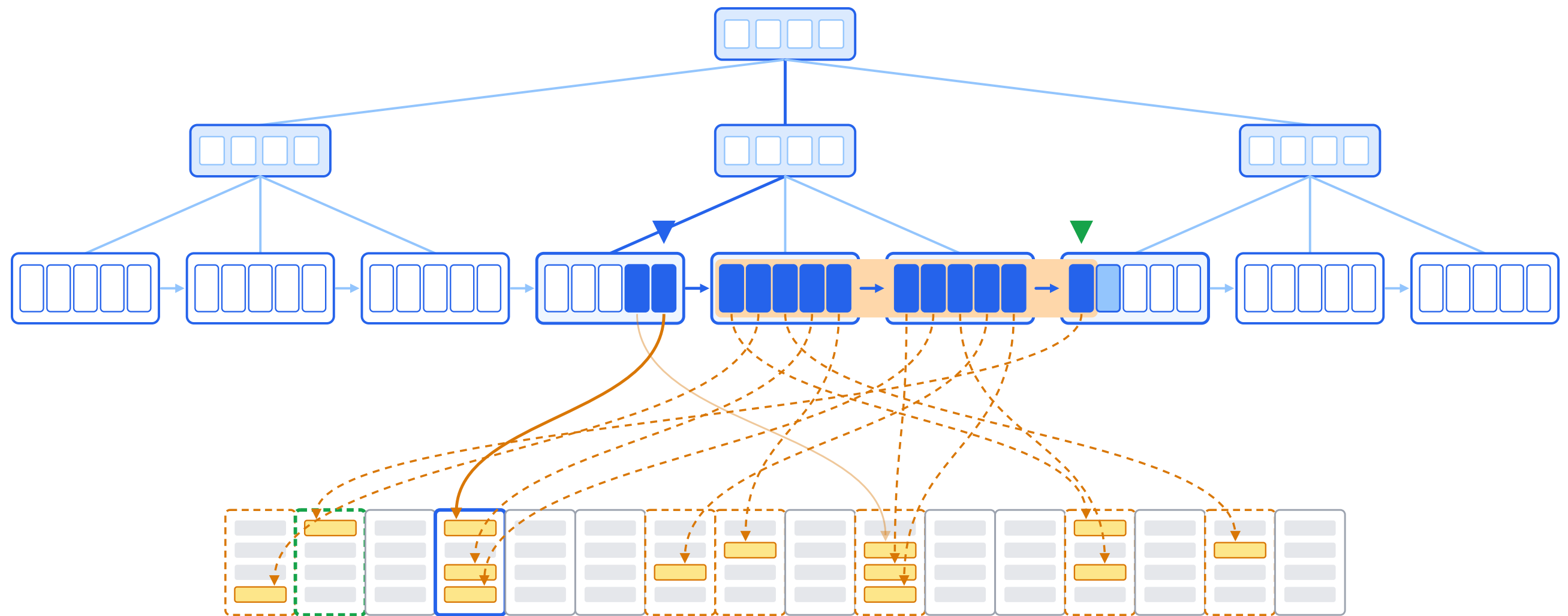
Plain index scans: prefetching

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**

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 by the examples I'll show
- 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)

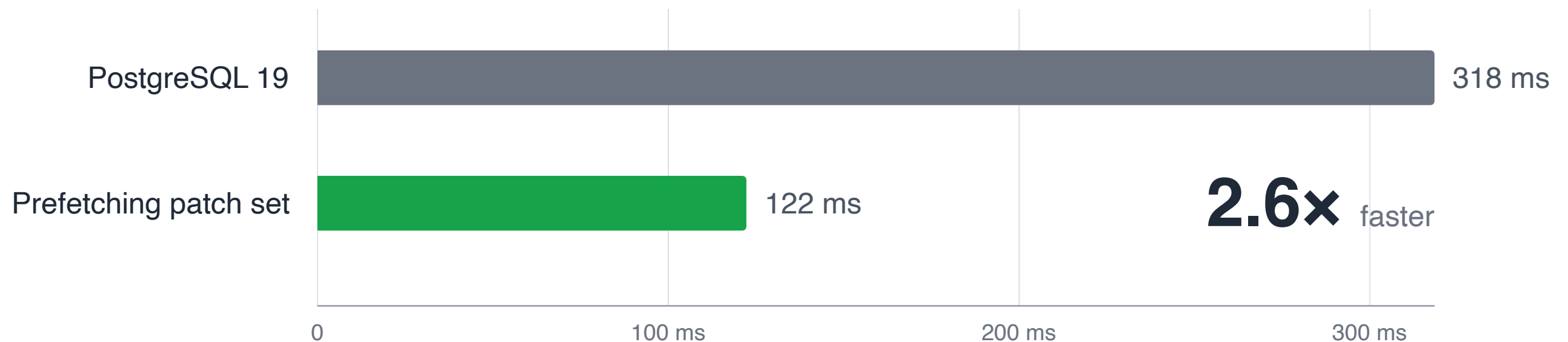


-- Sequential forwards scan

SELECT * FROM t

WHERE a BETWEEN 16336 AND 49103

ORDER BY a;

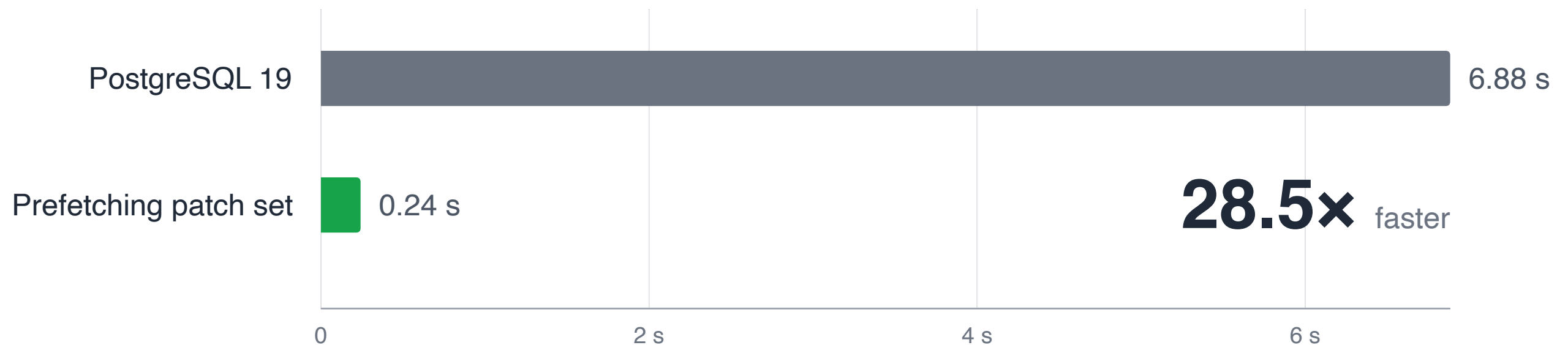


-- Sequential backwards scan

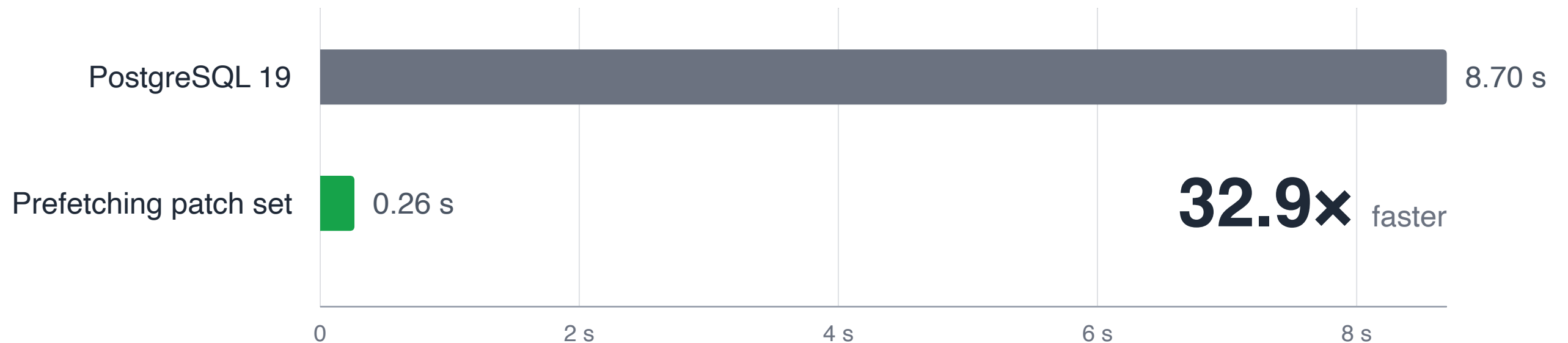
SELECT * FROM t

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;
```



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

Makes plain index scans far more competitive with bitmap scans, while preserving their natural advantages (ordered scan output, etc)

