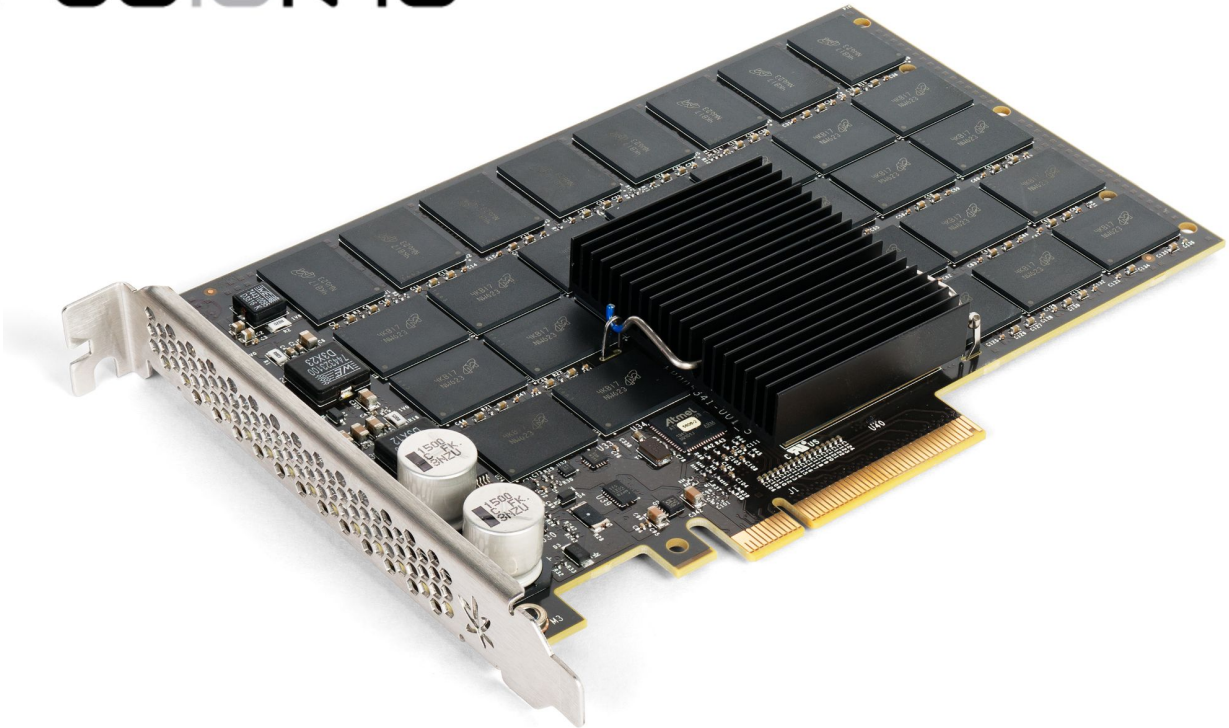
Indexing Ethereum

Best-in-class PCIe Gen5 performance[1,2]

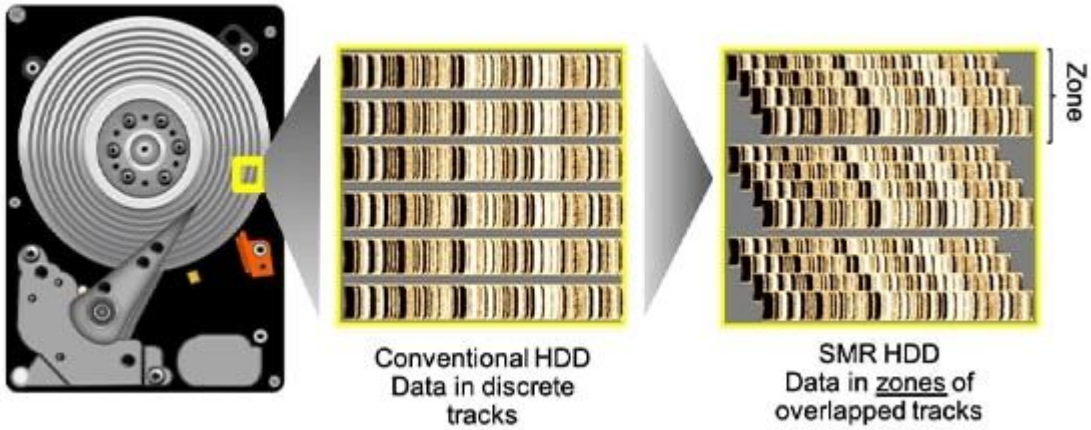
The Micron 6550 ION SSD delivers the industry-leading, high-capacity SSD performance needed for AI workloads. With 12.0 GB/s sequential reads and 5.0 GB/s sequential writes at only 20 watts, it provides up to 179% better performance than competitors' products, with up to a 20% power savings. With its exceptional write performance, it can write 61.44TB in 3.4 hours, which is up to 150% faster than the competition.[7]

FUSION-IO



Increasing Magic Pocket write throughput by removing our SSD cache disks

// By Ankur Kulshrestha, Rajat Goel, and Sandeep Ummadi • Dec 08, 2022



Gen 1	Gen 2	Gen 3	Gen 4	Gen 5	Gen 6
144 TB	192-240 TB	270 TB	624-720 TB	1.4-1.6 PB	1.8+ PB
					
Big Show	Andre	Atlas	Axel	Stimpy	Scooby
HDD capacity: 4 TB HDD count: 36	HDD capacity: 4 TB HDD count: 48-60	HDD capacity: 6-8 TB HDD count: 35-45	HDD capacity: 8 TB HDD count: 78-90	HDD capacity: 14-16 TB HDD count: 100-102	HDD capacity: 18+ TB HDD count: 100-102
2013	2014	2016	2017	2018	2021+

25 topics / 25 minutes = 1 topic / minute

1. ABI Decoding
2. JSON vs. Protobufs
3. getLogs
4. bloom filters
5. SSD Write Amplification
6. 100TB SMR HDD
7. Geth
8. Reth
9. Freezer Files
10. LevelDB
11. libmdbx
12. RLP
13. 32 byte bools
14. RIIR
15. LSM Trees
16. B+Trees
17. Patricia Merkle Tries
18. ZFS compression
19. TOAST
20. UNLOGGED TABLES
21. BRIN Indexing
22. REORGs
23. Minimizing WAL
24. GIN indexing
25. Is pg_dump a backup tool?

50 topics / 25 minutes = 1 topic per ½ minute

1. ABI Decoding
2. JSON vs. Protobufs
3. getLogs
4. bloom filters
5. SSD Write Amplification
6. 100TB SMR HDD
7. Geth
8. Reth
9. Freezer Files
10. LevelDB
11. libmdbx
12. RLP
13. 32 byte bools
14. RIIR
15. LSM Trees
16. B+Trees
17. Patricia Merkle Tries
18. ZFS compression
19. TOAST
20. UNLOGGED TABLES
21. BRIN Indexing
22. REORGs
23. Minimizing WAL
24. GIN indexing
25. Is pg_dump a backup tool?
26. COPY vs. INSERT
27. Table Partitions
28. Topic filters
29. Block range limit
30. EXPLAIN BUFFERS
31. shared_buffers
32. effective_cache_size
33. Flamegraphs
34. auto_explain
35. track_io_timing
36. CTIDS
37. Avoiding UPDATES
38. ETL vs. EL
39. Creating index after backfill
40. Index Organized Tables
41. Larry Ellison joke
42. CLUSTER
43. COVERING INDEX
44. BITMAP Heap Scan
45. SET application_name
46. Buffers hit as unit of work
47. EBS GP3 vs IO2
48. AWS vs. Hetzner
49. pg_stat_statements
50. log_min_duration

25 minutes / 3 topics = 8.33333… minutes per topic

1. How I think about dependencies
2. Essential Indexing Pattern
3. Postgres Tips

You Should Not Buy My Software

I sell indexing software.

But today I am going to make the case why you shouldn't buy it.

Part I - How I Think About Dependencies

Indexing Ethereum

Providing an API (ie SQL) to Ethereum's data optimized for Applications

Primitives

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER
EIP-150 REVISION (e39fbe8 - 2017-09-26)

DR. GAVIN WOOD
FOUNDER, ETHEREUM & ETHCORE
GAVIN@ETHCORE.IO

Abstract. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, not the least being Bitcoin. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

1. Introduction

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bitcoin have demonstrated, through the power of the default, consensus mechanisms and voluntary respect of the social contract that it is possible to use the internet to make a decentralised value-transfer system, shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up systems such as Namecoin adapted this original "currency application" of the technology into other applications albeit rather simplistic ones.

Ethereum is a project which attempts to build the generalised technology; technology on which all transaction-based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.

1.1. **Driving Factors.** There are many goals of this project; one key goal is to facilitate transactions between consenting individuals who would otherwise have no means to trust one another. This may be due to geographical separation, interfacing difficulty, or perhaps the incompatibility, incompetence, unwillingness, expense, uncertainty, inconvenience or corruption of existing legal systems. By specifying a state-change system through a rich and unambiguous language, and furthermore architecting a system such that we can reasonably expect that an agreement will be thus enforced autonomously, we can provide a means to this end.

Dealings in this proposed system would have several attributes not often found in the real world. The incorruptibility of judgement, often difficult to find, comes naturally from a disinterested algorithmic interpreter. Transparency, or being able to see exactly how a state or judgement came about through the transaction log and rules or instructional codes, never happens perfectly in human-based systems since natural language is necessarily vague, information is often lacking, and plain old prejudices are difficult to shake.

Overall, I wish to provide a system such that users can be guaranteed that no matter with which other individuals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

1.2. **Previous Work.** Buterin [2013a] first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a blockchain with a Turing-complete language and an effectively unlimited inter-transaction storage capability remains unchanged.

Dwork and Naor [1992] provided the first work into the usage of a cryptographic proof of computational expenditure ("proof-of-work") as a means of transmitting a value signal over the Internet. The value-signal was utilised here as a spam deterrence mechanism rather than any kind of currency, but critically demonstrated the potential for a basic data channel to carry a *strong economic signal*, allowing a receiver to make a physical assertion without having to rely upon *trust*. Back [2002] later produced a system in a similar vein.

The first example of utilising the proof-of-work as a strong economic signal to secure a currency was by Vishnumurthy et al. [2003]. In this instance, the token was used to keep peer-to-peer file trading in check, providing "consumers" with the ability to make micro-payments to "suppliers" for their services. The security model afforded by the proof-of-work was augmented with digital signatures and a ledger in order to ensure that the historical record couldn't be corrupted and that malicious actors could not spoof payment or unjustly complain about service delivery. Five years later, Nakamoto [2008] introduced another such proof-of-work-secured value token, somewhat wider in scope. The fruits of this project, Bitcoin, became the first widely adopted global decentralised transaction ledger.

Other projects built on Bitcoin's success; the alt-coins introduced numerous other currencies through alteration to the protocol. Some of the best known are Litecoin and Primecoin, discussed by Sprankel [2013]. Other projects sought to take the core value content mechanism of the

Primitives

Block, Header, Transaction, Receipt, Log, State are stored in a database in the EC and those databases are optimized for executing Ethereum Transactions.

Not necessarily a good format for an API

Has anyone used a freezer file?

Applications want access to decoded (abi) logs and transactions

We also want to create **database indexes** over this data for fast access

Dependency Landscape

No Dependencies: Run your own node

- Pro: Private. No lock-in
- Con: Needs: sys. admin and DBA

No Primitives: Authenticated API access to a data provider

- Pro: No sys. admin or indexer program
- Con: Lock-in

Hybrid: Access hosted node via the standard JSON RPC API

- Pro: Commoditized API market (minimal lock-in)
- Con: DBA and indexer program

Build vs Buy

High Value Project

- Low Skills
  - **Buy**. Try to find something everyone else is using.
  - You probably won't get fired for picking the average.
  - **Try to hire or grow your skills.**
    - ***It can absolutely be done. Not rocket science!***
- High Skills
  - **Build.** If it's critical to your company you probably should own the data.
  - At the very least don't outsource a core competency or worse:
  - Don't outsource your thinking.

Build vs Buy

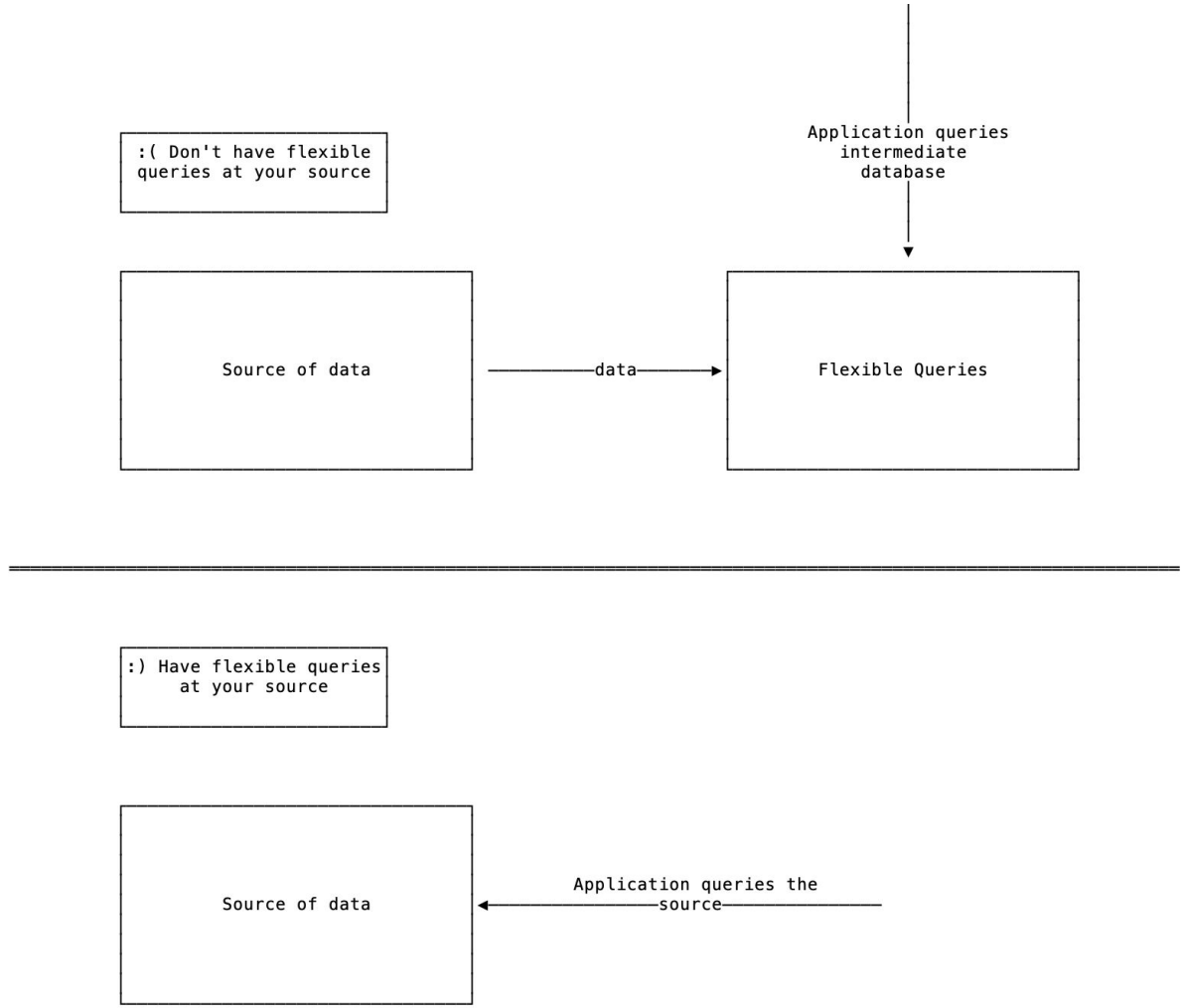
Low Value Project

- Low Skills
  - **Buy**. Jam stuff together until it works.
- High Skills
  - **Both can work!**
  - **Build.** Maybe you have a box of tools that can easily get'r done
  - **Buy.** Someone solved the problem exactly how you would have solved it.
  - You didn't outsource your thinking, you outsourced the maintenance!

Part II - Essential Indexing Pattern

```
create table if not exists blocks (
  num bigint,
  hash bytea,
  primary key(num)
);
create table if not exists foo (
  block_num bigint,
  bar bytea
);
```

```
async index() {
  const client = await this.pg.connect();
  await client.query("begin");
  const remote = await this.remoteLatest();
  const local = await this.localLatest(client);
  if (local.num ⩾ remote.num) {
    // nothing to do
    await client.query("rollback");
    return;
  }
  const batchSize = 100n;
  const delta = min(remote.num - local.num, batchSize);
  const [from, to] = [
    await this.getBlock(local.num + 1n),
    await this.getBlock(local.num + delta),
  ];
  if (from.parent ≠ local.hash) {
    // reorg
    await client.query("delete from blocks where num ⩾ $1", [local.num]);
    await client.query("delete from foo where block_num ⩾ $1", [local.num]);
    await client.query("commit");
    return;
  }
  const logs = await this.getLogs(from.num, to.num, {});
  streamLogs(logs).pipe(
    copy("COPY logs (block_num, bar) FROM STDIN WITH (FORMAT binary)"),
  );
  await client.query("insert into blocks(num, hash) values ($1, $2)", [
    to.num,
    to.hash,
  ]);
  await client.query("commit");
}
```


:(Don't have flexible queries at your source
Application queries intermediate database
Source of data
data
Flexible Queries
:) Have flexible queries at your source
Source of data
Application queries the source

Part III - Postgres Tips

Postgres Tips

COPY

- Doesn't parse SQL
- Optimized IO path

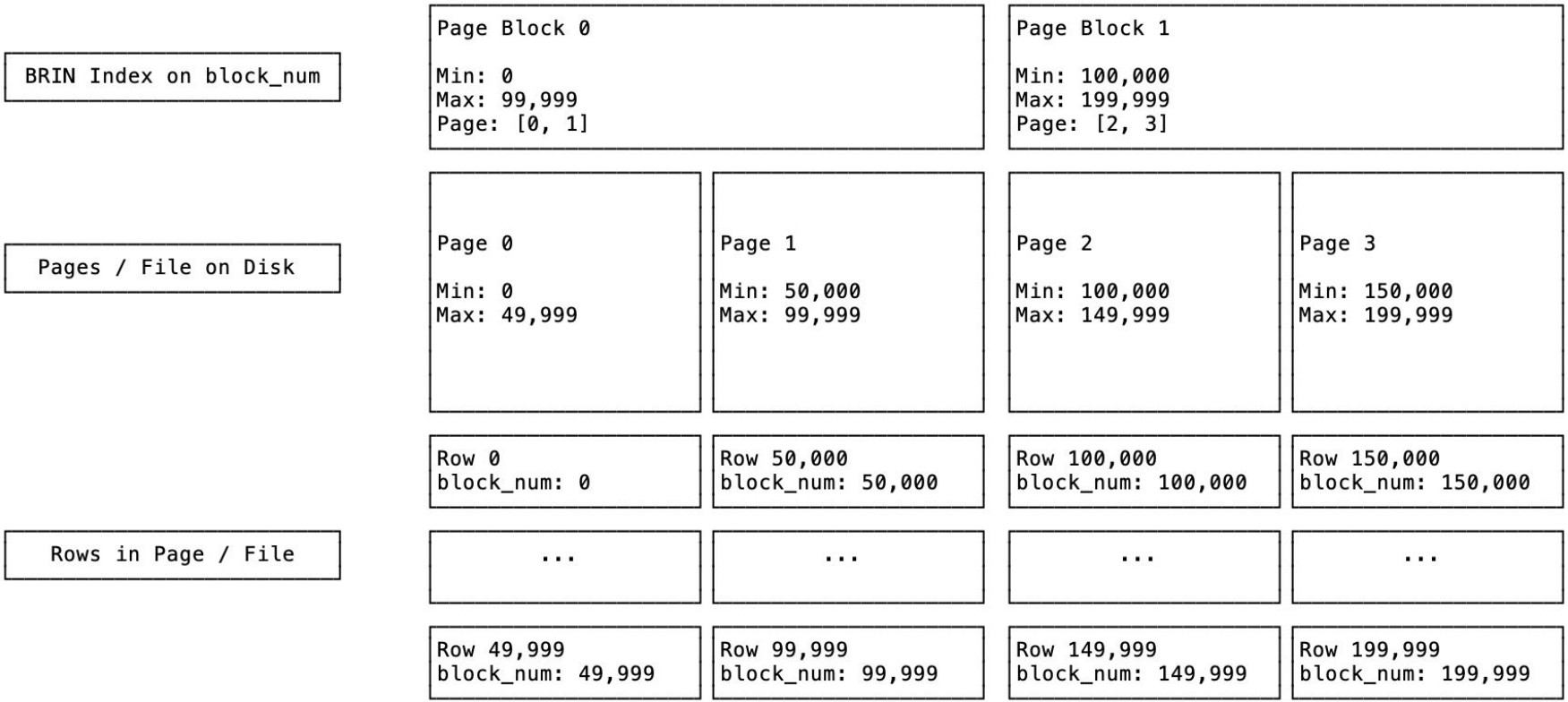
Advisory Locks

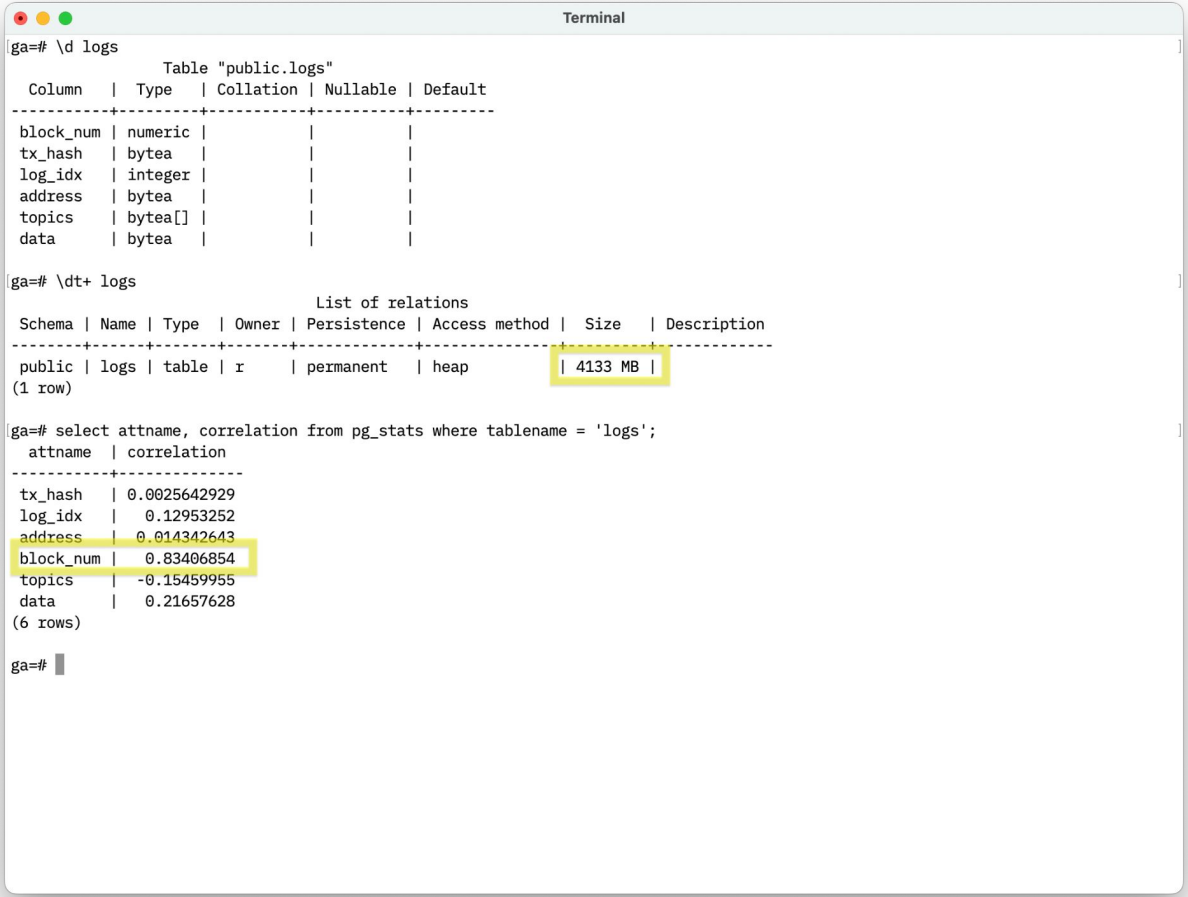
- Easy way to ensure only 1 process is indexing at a time

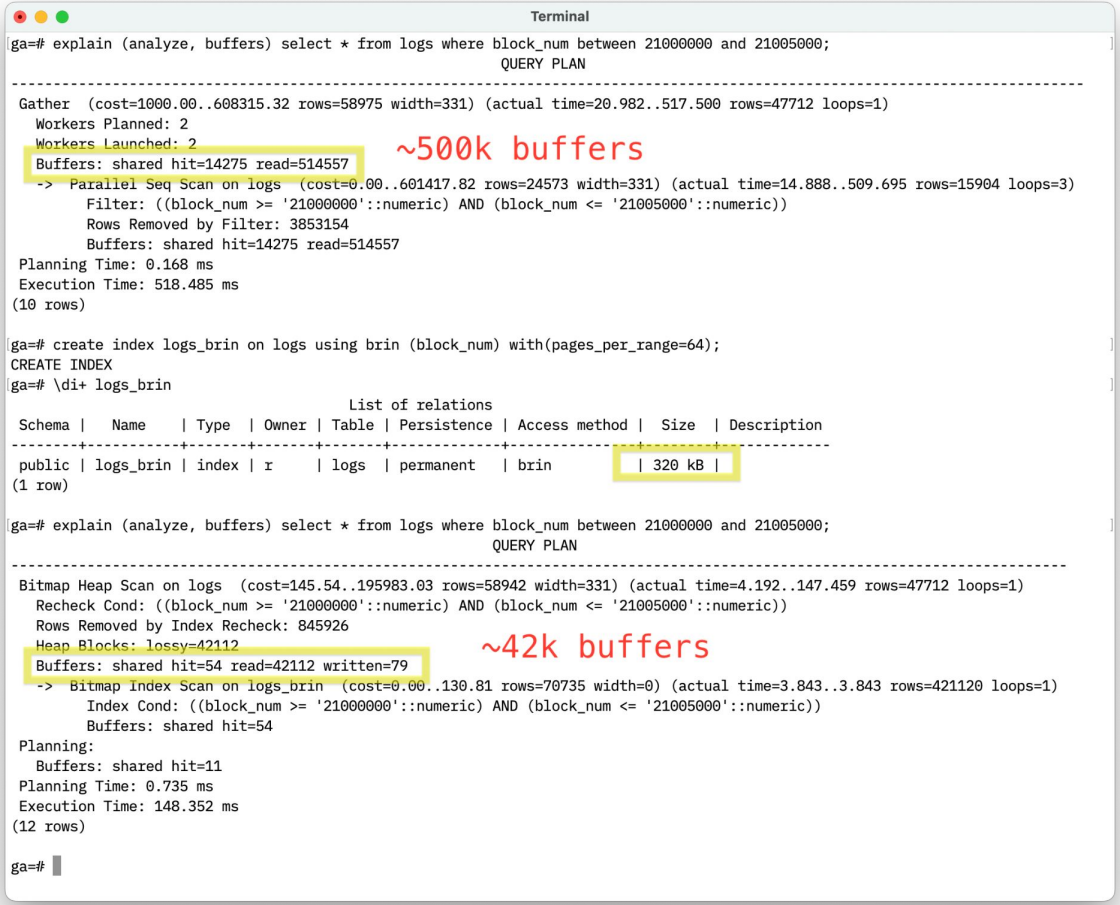
Table Partitioning

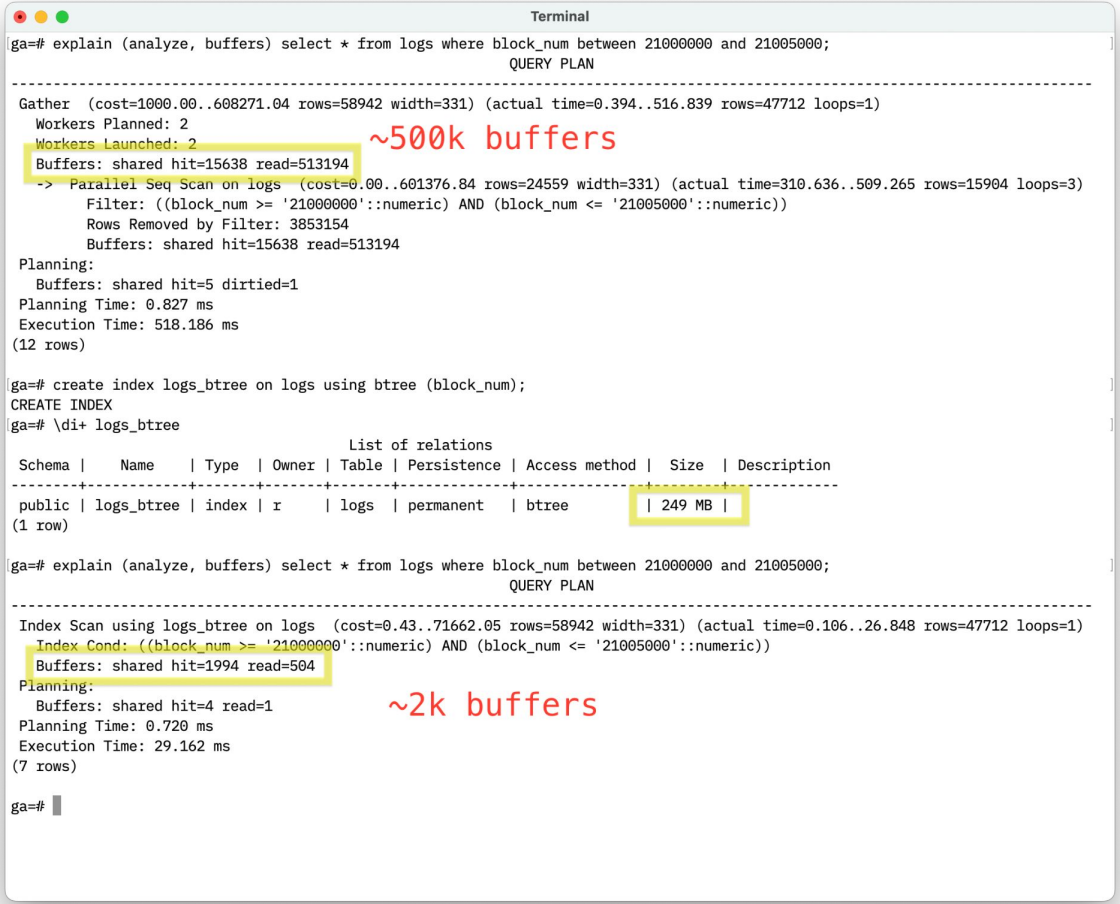
- Transparently improves table and index scans when you can't index
- Partition on contract address
  - Some partitions will have more data than others. Feature?

BRIN


BRIN Index on block_num
Page Block 0
Min: 0
Max: 99,999
Page: [0, 1]
Page Block 1
Min: 100,000
Max: 199,999
Page: [2, 3]
Pages / File on Disk
Page 0
Min: 0
Max: 49,999
Page 1
Min: 50,000
Max: 99,999
Page 2
Min: 100,000
Max: 149,999
Page 3
Min: 150,000
Max: 199,999
Rows in Page / File
Row 0
block_num: 0
Row 50,000
block_num: 50,000
Row 100,000
block_num: 100,000
Row 150,000
block_num: 150,000
...
...
...
...
Row 49,999
block_num: 49,999
Row 99,999
block_num: 99,999
Row 149,999
block_num: 149,999
Row 199,999
block_num: 199,999


Terminal
ga=# \d logs
Table "public.logs"
Column | Type | Collation | Nullable | Default
block_num | numeric |
tx_hash | bytea |
log_idx | integer |
address | bytea |
topics | bytea[] |
data | bytea |
ga=# \dt+ logs
List of relations
Schema | Name | Type | Owner | Persistence | Access method | Size | Description
public | logs | table | r | permanent | heap | 4133 MB |
(1 row)
ga=# select attname, correlation from pg_stats where tablename = 'logs';
attname | correlation
tx_hash | 0.0025642929
log_idx | 0.12953252
address | 0.014342643
block_num | 0.83406854
topics | -0.15459955
data | 0.21657628
(6 rows)
ga=#


Terminal
ga=# explain (analyze, buffers) select * from logs where block_num between 21000000 and 21005000;
QUERY PLAN
Gather (cost=1000.00..608315.32 rows=58975 width=331) (actual time=20.982..517.500 rows=47712 loops=1)
Workers Planned: 2
Workers Launched: 2
Buffers: shared hit=14275 read=514557
~500k buffers
-> Parallel Seq Scan on logs (cost=0.00..601417.82 rows=24573 width=331) (actual time=14.888..509.695 rows=15904 loops=3)
Filter: ((block_num >= '21000000'::numeric) AND (block_num <= '21005000'::numeric))
Rows Removed by Filter: 3853154
Buffers: shared hit=14275 read=514557
Planning Time: 0.168 ms
Execution Time: 518.485 ms
(10 rows)
ga=# create index logs_brin on logs using brin (block_num) with(pages_per_range=64);
CREATE INDEX
ga=# \di+ logs_brin
List of relations
Schema | Name | Type | Owner | Table | Persistence | Access method | Size | Description
public | logs_brin | index | r | logs | permanent | brin | 320 kB |
(1 row)
ga=# explain (analyze, buffers) select * from logs where block_num between 21000000 and 21005000;
QUERY PLAN
Bitmap Heap Scan on logs (cost=145.54..195983.03 rows=58942 width=331) (actual time=4.192..147.459 rows=47712 loops=1)
Recheck Cond: ((block_num >= '21000000'::numeric) AND (block_num <= '21005000'::numeric))
Rows Removed by Index Recheck: 845926
Heap Blocks: lossy=42112
Buffers: shared hit=54 read=42112 written=79
~42k buffers
-> Bitmap Index Scan on logs_brin (cost=0.00..130.81 rows=70735 width=0) (actual time=3.843..3.843 rows=421120 loops=1)
Index Cond: ((block_num >= '21000000'::numeric) AND (block_num <= '21005000'::numeric))
Buffers: shared hit=54
Planning:
Buffers: shared hit=11
Planning Time: 0.735 ms
Execution Time: 148.352 ms
(12 rows)
ga=#


Terminal
ga=# explain (analyze, buffers) select * from logs where block_num between 21000000 and 21005000;
QUERY PLAN
Gather (cost=1000.00..608271.04 rows=58942 width=331) (actual time=0.394..516.839 rows=47712 loops=1)
Workers Planned: 2
Workers Launched: 2
Buffers: shared hit=15638 read=513194
-> Parallel Seq Scan on logs (cost=0.00..601376.84 rows=24559 width=331) (actual time=310.636..509.265 rows=15904 loops=3)
Filter: ((block_num >= '21000000'::numeric) AND (block_num <= '21005000'::numeric))
Rows Removed by Filter: 3853154
Buffers: shared hit=15638 read=513194
Planning:
Buffers: shared hit=5 dirtied=1
Planning Time: 0.827 ms
Execution Time: 518.186 ms
(12 rows)
~500k buffers
ga=# create index logs_btree on logs using btree (block_num);
CREATE INDEX
ga=# \di+ logs_btree
List of relations
Schema | Name | Type | Owner | Table | Persistence | Access method | Size | Description
public | logs_btree | index | r | logs | permanent | btree | 249 MB |
(1 row)
ga=# explain (analyze, buffers) select * from logs where block_num between 21000000 and 21005000;
QUERY PLAN
Index Scan using logs_btree on logs (cost=0.43..71662.05 rows=58942 width=331) (actual time=0.106..26.848 rows=47712 loops=1)
Index Cond: ((block_num >= '21000000'::numeric) AND (block_num <= '21005000'::numeric))
Buffers: shared hit=1994 read=504
Planning:
Buffers: shared hit=4 read=1
Planning Time: 0.720 ms
Execution Time: 29.162 ms
(7 rows)
~2k buffers
ga=#

67.4. Implementation

This section covers B-Tree index implementation details that may be of use to advanced users. See `src/backend/access/nbtree/README` in the source distribution for a much more detailed, internals-focused description of the B-Tree implementation.

67.4.1. B-Tree Structure

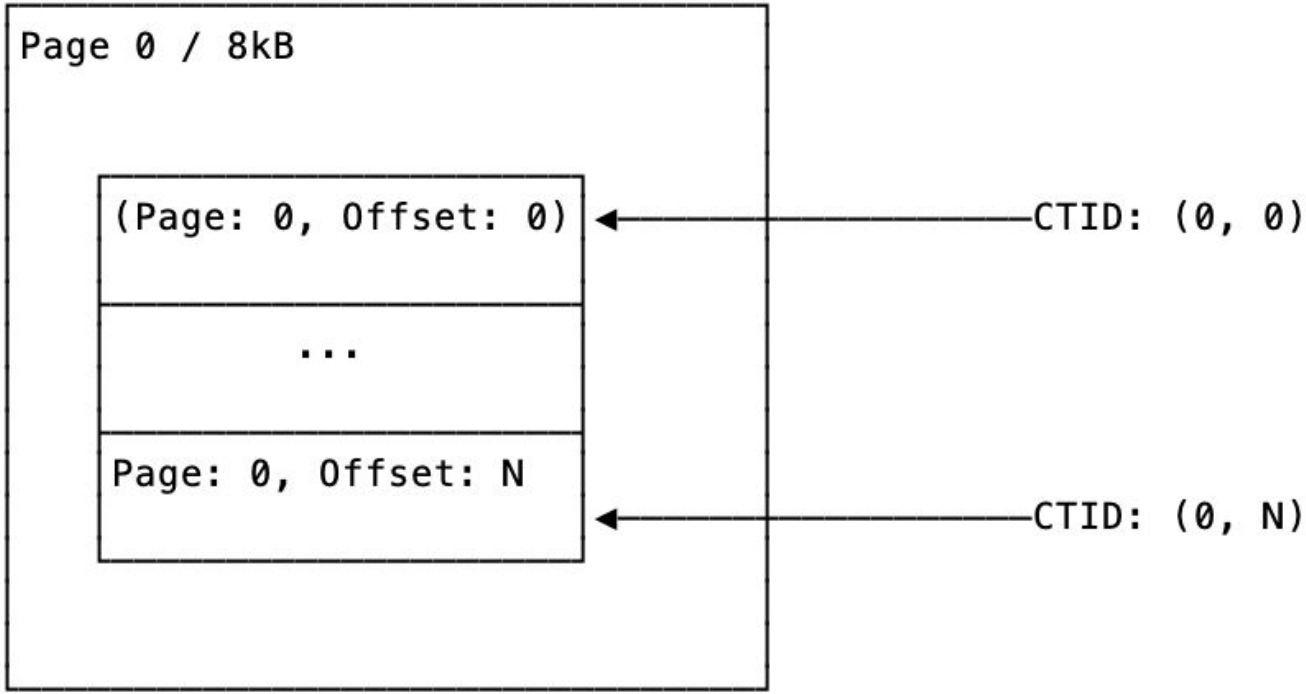
PostgreSQL B-Tree indexes are multi-level tree structures, where each level of the tree can be used as a doubly-linked list of pages. A single metapage is stored in a fixed position at the start of the first segment file of the index. All other pages are either leaf pages or internal pages. Leaf pages are the pages on the lowest level of the tree. All other levels consist of internal pages. Each leaf page contains tuples that point to table rows. Each internal page contains tuples that point to the next level down in the tree. Typically, over 99% of all pages are leaf pages. Both internal pages and leaf pages use the standard page format described in Section 73.6.

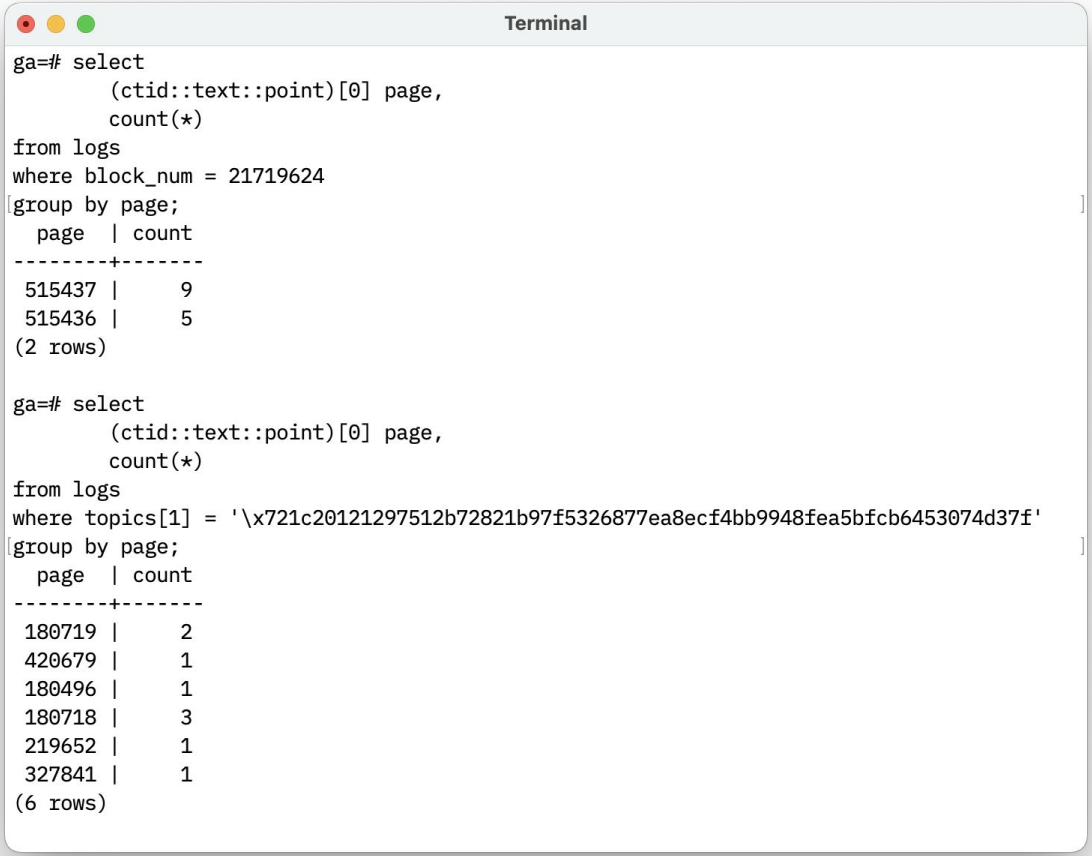
New leaf pages are added to a B-Tree index when an existing leaf page cannot fit an incoming tuple. A *page split* operation makes room for items that originally belonged on the overflowing page by moving a portion of the items to a new page. Page splits must also insert a new *downlink* to the new page in the parent page, which may cause the parent to split in turn. Page splits "cascade upwards" in a recursive fashion. When the root page finally cannot fit a new downlink, a *root page split* operation takes place. This adds a new level to the tree structure by creating a new root page that is one level above the original root page.

67.4.2. Bottom-up Index Deletion

B-Tree indexes are not directly aware that under MVCC, there might be multiple extant versions of the same

CTIDs




Terminal
ga=# select
(ctid::text::point)[0] page,
count(*)
from logs
where block_num = 21719624
group by page;
page | count
--------+-------
515437 | 9
515436 | 5
(2 rows)
ga=# select
(ctid::text::point)[0] page,
count(*)
from logs
where topics[1] = '\x721c20121297512b72821b97f5326877ea8ecf4bb9948fea5bfcb6453074d37f'
group by page;
page | count
--------+-------
180719 | 2
420679 | 1
180496 | 1
180718 | 3
219652 | 1
327841 | 1
(6 rows)

Terminal

```
ga=# cluster logs using logs_topics_idx;
CLUSTER
ga=# select
        (ctid::text::point)[0] page,
        count(*)
from logs
where topics[1] = '\x721c20121297512b72821b97f5326877ea8ecf4bb9948fea5bfcb6453074d37f'
group by page;
  page   | count
--------+-------
 207448 |     1
 207447 |     8
(2 rows)

ga=# select
        (ctid::text::point)[0] page,
        count(*)
from logs
where block_num = 21719624
group by page;
  page   | count
--------+-------
 526949 |     6
 316808 |     2
 207071 |     1
 297610 |     1
 366991 |     2
 286056 |     1
 421358 |     1
(7 rows)
```

@ryansmith

/pg