NEON

Taking Postgres to the 21st Century

Database Should Just Be a URL

Nikita Shagmunov bio

SQL Server 2005-2010

- XML, Geo Spatial, .NET, Typesystem

Singlestore 2011 - 2020

- Founder CTO
- Then CEO

Neon 2021 - now

- Founder CEO

Why Database Should Just be a URL?

Developer Experience

- Instant everything
- Tune nothing
- Branching
- Consumption based pricing

Shareability

- Share a URL
- Publish to the web

Access from anywhere

- Datacenter
- Edge
- Browser

Postgres Early days

Postgres left Berkeley in 1996

Foundation was already there:

- Extendable types and functions
- SQL support, cost-based optimizer
- Multiple index AMs: b-tree, hash
- Storage manager interface, md.c and mm.c
- VACUUM 🤯
    - In database early design decisions stay for a LONG time

PostgreSQL Storage History

1990-2000: MVCC, VACUUM, extendable index types

2001: WAL, TOAST (blob store)
2005: Subtransactions, online backup, PITR, tablespaces
2010: pg_upgrade, streaming replication, hot standby

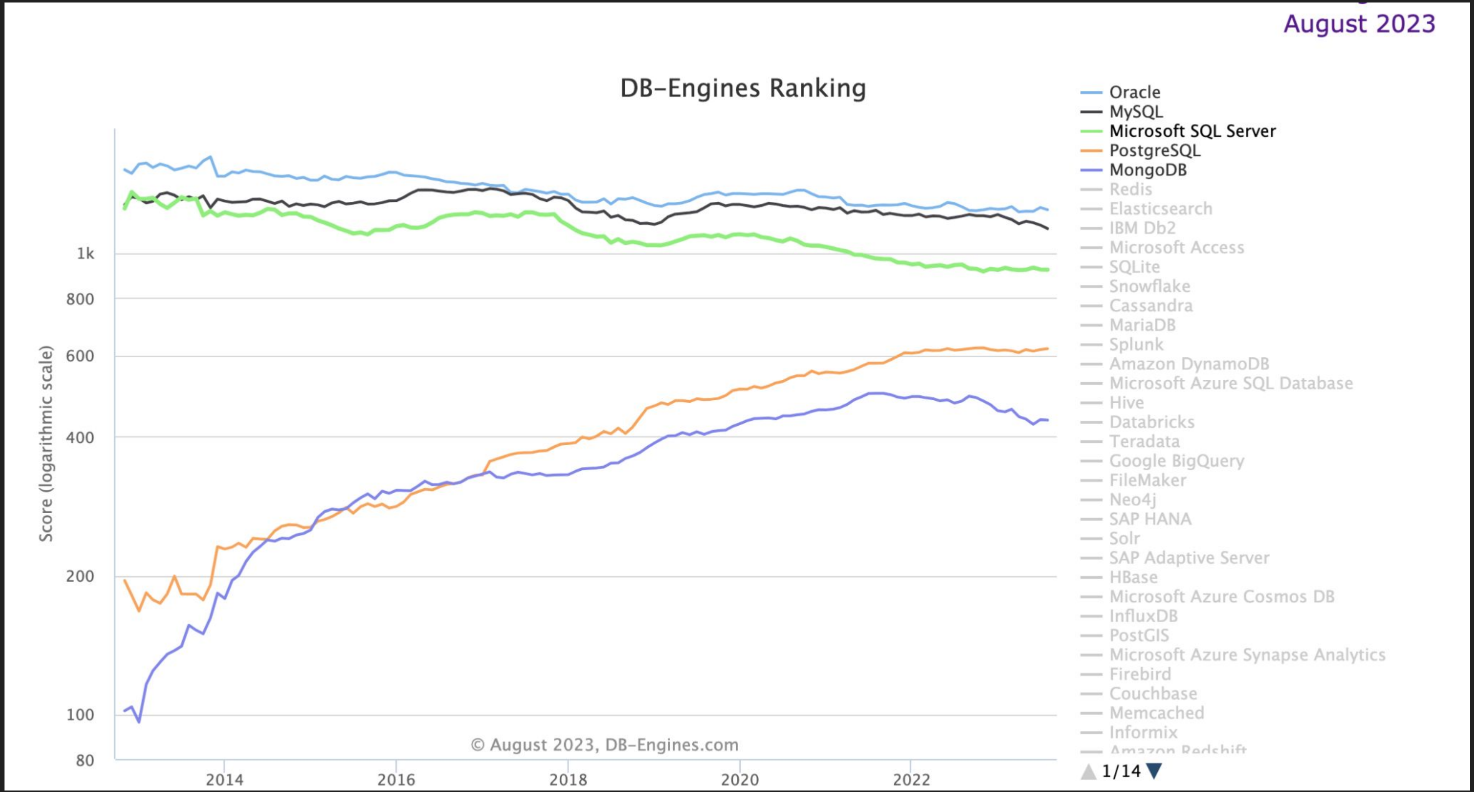
2013: FDWs
2017: Logical replication
2018: Partitioning, JIT compilation

Storage Architecture

By 2010-2015, Postgres arrived at a “traditional” storage architecture:

- Page based system. No redo log
- Primary and 1-2 hot standbys
- Streaming replication
- Scripting to manage the cluster, e.g. Patroni
- Backups and WAL archiving to cloud storage

Where is Postgres now?



The Arrival of the Cloud

1st generation cloud architecture

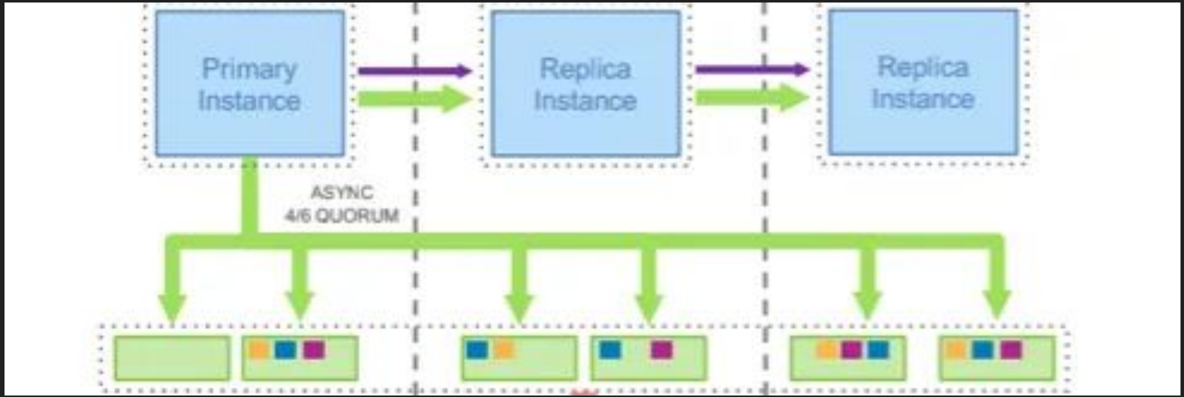
Lift and shift:

- Servers to cloud instances
- Data is stored on expensive EBS volumes
- Fast io achieved by provisioned IOPs
- Backups to cloud storage

2nd Generation Cloud Architecture (AWS Aurora)

Separation of storage and compute

- Write 6 copies across 3 AZs
- Use gossip protocol for persistence



AWS Aurora Wins Sigmod Systems Award

SIGMOD AWARDS

Amazon Aurora

The Aurora Database System

2019 SIGMOD Systems Award

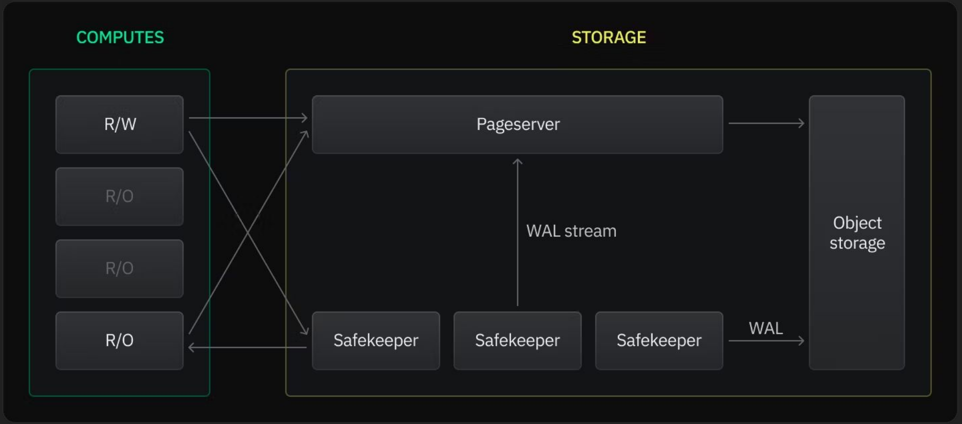
3rd Generation Cloud Architecture

Built from the ground up in Rust. Open source with Apache 2.0 license

- Safekeepers + S3 is the source of truth
- Pageserver is a persistent cache
    - Get access any page by (page_id, lsn, tenant_id)

Benefits:

- Immediate startup of a new compute node
    - No WAL replay
    - No checkpointing
- Immediate start of the read replica
- *At any point-in-time (LSN)*
- Branching



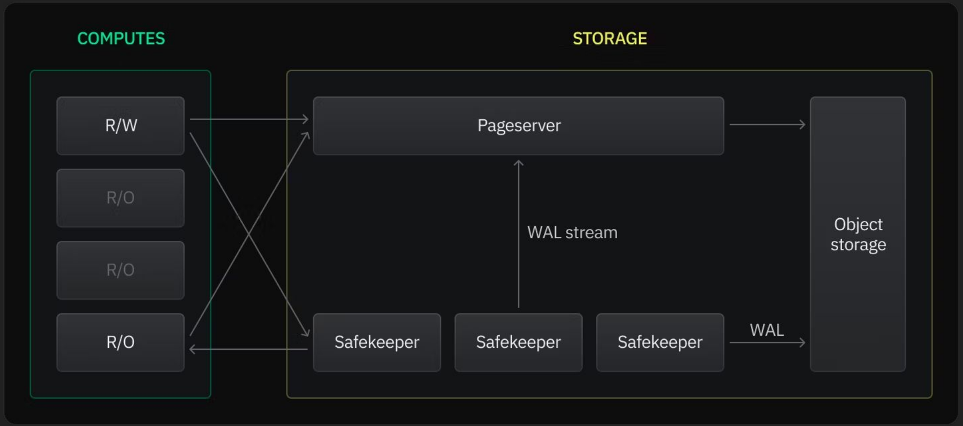
Storage

Safekeepers

- Simplified Paxos protocol
- Immediately asynchronous stream to S3
- Stream WAL to page servers and read replica

Benefits:

- System of record is safekeepers and S3
- Full region catastrophic failure allows to recover from s3
- Read replicas are instant



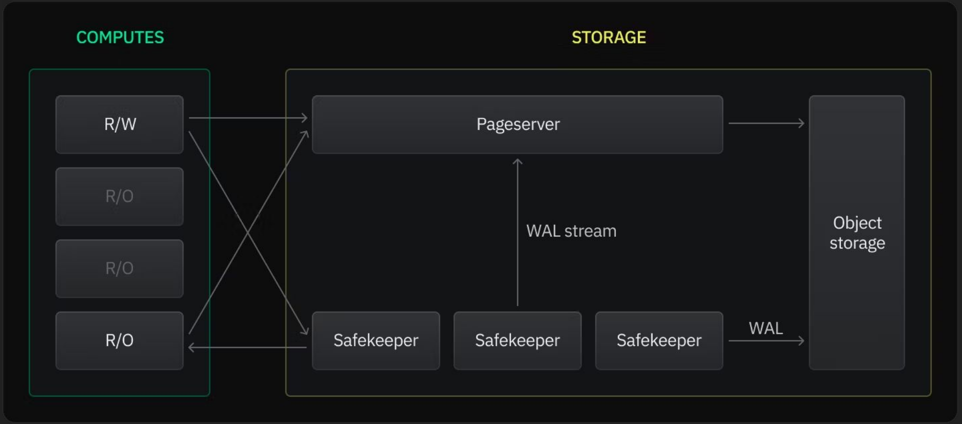
Storage

Pageservers

- Data is organized in “temporal” LSM trees
- Supports GetPage@LSN access method
- Support LSM layer offload to S3

Benefits:

- Enables time machine and branching
- Cost effective due to memory hierarchy
- Constant battle with write amplification



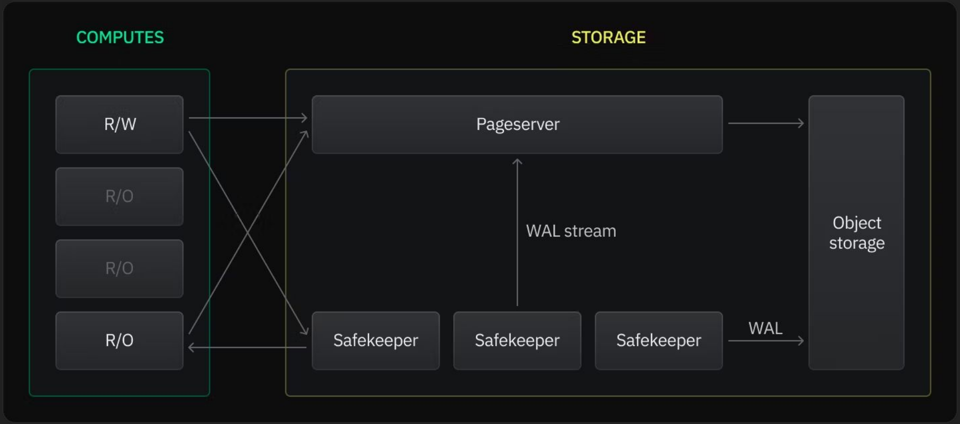
Storage

Pageservers

- Data is organized in “temporal” LSM trees
- Supports GetPage@LSN access method
- Support LSM layer offload to S3

Benefits:

- Enables time machine and branching
- Cost effective due to memory hierarchy
- Constant battle with write amplification



Serverless Goals (database is a URL)

Automatic scaling

- Down to zero
- Up to max host size
  - Plus read replicas
  - Across the globe

Challenges

- Replication lag between read replicas
- Cache coherency across read replicas
- Multi-master implementation to scale out writes
  - Need to row based WAL for multi-master

Scaling compute. General Architecture

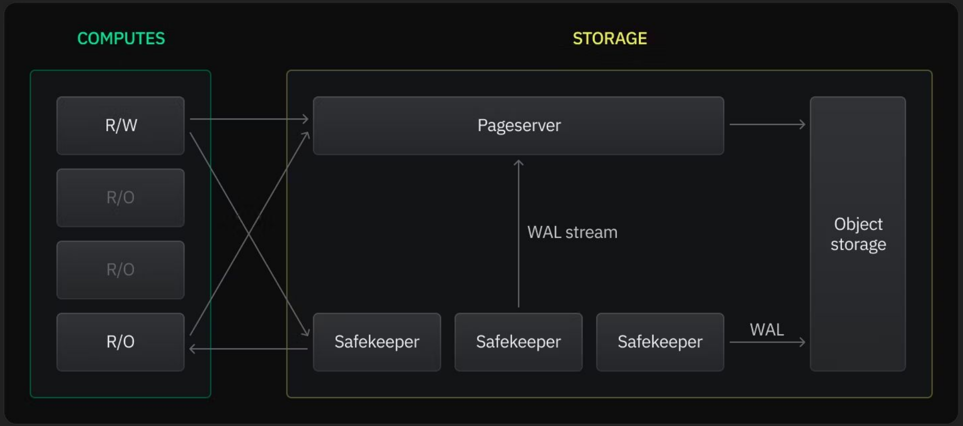
Each compute is a VM:

- QEMU+kvm with actual CPU+memory hotplug

Using k8s as our orchestration layer

- We build NeonVM - k8s support for VMs + custom scheduler

All code in the open sourced and we develop in the open



Scaling VMs Within One Host

- Using CPU and memory hotplug - building on the shoulders of giants
  - We’ll scale you down, too!
- Scaling decisions use load average & memory usage
- Inside the VM, react to cgroup v2 memory pressure notifications
  - Faster reactions than we could ever get with polling
- Custom resizable postgres cache
  - 1MB chunk size so static overhead is small

Scaling computes: Rebalancing with live migrations

Allows moving VMs between physical hosts without restart:

- Copy disk + memory, pause, copy CPU states, resume on the new host
- Minimal (100ms) interruption during hand-off

Prioritizes smaller & less active computes

- Even with scale-to-zero and autoscaling, people pay for mostly-idle DBs
- Migrations limited by network speed — smaller is faster

Easier instance scale-down / decommissioning K8s nodes

- No need to restart DBs
- Better uptime for users, easier to manage for us

Not all easy: Disk usage is harder to manage

- Can’t scale while migrating
- Disk usage means more data to move - migration takes longer

Scaling computes: K8s customizations

Represent VMs as a custom resource (CRD) in k8s:

- NeonVM is our abstraction layer

Custom k8s scheduler via plugins framework:

- Knows about VMs, active rebalancing between nodes
- Approves scaling requests to avoid overcommitting memory (OOM is not good!)

Everything needs to know about VMs

- Also modified cluster-autoscaler

Scaling computes: The hard parts

We're hitting uncommon code paths:

- Issues with scaling failures and OOMs
- Kernel panics due to tmpfs usage

Uncomfortable trade-offs:

- Disk scaling is hard to implement but valuable for caching
- Sometimes unavoidable — postgres spills to disk

Key postgres settings aren't built to dynamically scale:

- Built our own shared_buffers replacement because overhead from hashmap was too big
    - Caching is critical because fetching from storage is over the network

Tons of integration work needed to polish things:

- Index building
- Query planning

PostgreSQL connection scaling

A connection has a lot of state:

- Cursors, prepared statements, temporary tables, per-connection settings, metadata caches etc.
- Max # of connections is fixed at startup
- Memory management: fixed size areas for buffer pool and other things

These problems are exacerbated when you try to scale on the fly

- Switching from multi-process to multi-threaded architecture would help

Traditional solution is to use a connection pool:

- Helps with connections scaling
- Doesn’t address scaling memory
- Breaks lots of connection state

What is a connection anyway?

```
import { neon } from '@neondatabase/serverless';
const sql = neon(process.env.DATABASE_URL);

const [post] = await sql`SELECT * FROM posts WHERE id = ${postId}`;
// `post` is now { id: 12, title: 'My post', ... } (or undefined)
```

People building new stateless applications don’t think that way

- A connection pool? A cursor? What is that?
- Database is just a URL

Serverless driver

- Access to the database over HTTP
- No need to establish a connection

Postgres extendability

1995: B-tree, hash

1996: GiST: r-tree

2001: Postgis

2006: GIN: generalized inverted index, full text search

2014: jsonb, indexing

2015: BRIN: block range indexes

2019: Table access method interface

2021+: ???

2021: Vector embeddings

Supporting vectors

- Array data type is already there

Supporting KNN (k-nearest neighbour)

- Already there with full table scan

Supporting indexing for fast ANN

- Pgvector and pg_embedding
- IVFFlat and HNSW indexes

Future

- Sparse vectors
- Quantization
- Vamana indexes

What's Next

Don't bet against Postgres

Serverless will come to all platforms

Threaded model

- Better runs on windows
- Better parallel execution
- Easier to autoscale

No More Vacuum?

Postgres platform

- Extension package manager. We need something like NPM. Potentially http://pgt.dev

Thank you!

@neondatabase
@nikitabase
nikita@neon.tech