A TOUR OF AN INSERT

How we built our Postgres database in Rust

NEON

https://neon.tech
https://conradludgate.com

Speaker notes

At Neon, we have just officially launched our Serverless Postgres platform. At the heart is the same familiar C postgres implementation, but much of our special sauce is in our Rust infrastructure surrounding Postgres. Let's take a tour of an INSERT query and see how it all fits together.

```
psql "postgres://conrad:password@my-endpoint.ap-southeast-2.aws.neon.tech/main"
  -c "INSERT INTO rust_meetups (name, date) VALUES ('RustAU Sydney', '2024-04-17')"
```

Speaker notes

Let's take this simple query.

proxy / handshake

```rust
let (mut stream, params) = match handshake(stream, tls).await? {
    HandshakeData::Startup(stream, params) => (stream, params),
    HandshakeData::Cancel(cancel_key_data) => {
        return Ok(cancellation_handler
            .cancel_session(cancel_key_data, ctx.session_id)
            .await
            .map(|()| None)?)
    }
};
```

Speaker notes

First things first, we accept a new TCP connection and perform the postgres handshake. This sets up the TLS connection and also receives any startup params like user and database name.

proxy / authenticate

```rust
let res = user_info.authenticate(
    ctx,
    &mut stream,
    &config.authentication_config
).await;
let user_info = match res {
    Ok(auth_result) => auth_result,
    Err(e) => {
        return stream.throw_error(e).await?;
    }
};
```

Speaker notes

Now, we need to confirm you are who you say you are. We perform the SCRAM authentication protocol to securely verify your password is correct, with included MitM protection

proxy / wake

```rust
loop {
    match handle_try_wake(api.wake_compute(ctx).await) {
        // do not retry error
        Err(e) => {
            report_error(&e, false);
            return Err(e);
        }
        // retry error
        Ok(Continue(e)) => report_error(&e, true),
        // success
        Ok(Break(n)) => return Ok(n),
    }
}
```

Speaker notes

Neon is a serverless postgres platform. That means the servers are our problem. If your postgres server is not running, we start it now, making sure to handle any spurious or temporary failure modes.

proxy / passthrough

```rust
let mut node = connect(ctx, &params, &user_info).await?;
let session = cancellation_handler.get_session();
prepare_client(&node, &session, &mut stream).await?;
proxy_pass(stream, node, session).await?;
```

Speaker notes

You are authenticated, your database is running. Connect to the database and switch to the transparent passthrough mode.

pageserver / GetPage@LSN

```
let neon_fe_msg = PagestreamFeMessage::parse(
    &mut copy_data_bytes.reader()
)?;
let response = match neon_fe_msg {
    PagestreamFeMessage::GetPage(req) => {
        self.handle_get_page_at_lsn_request(
            tenant_id, timeline_id, &req, &ctx
        ).await
    }
}
```

Speaker notes

Proxy has streamed the insert query to postgres, now postgres needs to load the table schema and any index data for BTree operations. Neon has separated the storage layer out of postgres. This is what allows us to have 100ms cold starts as your database VM does not need to download the potentially several GB base backup to startup. Instead, any non-cached pages will skip the disk and go straight to our custom storage engine, called the Pageserver.

pageserver / GetPage@LSN / lookup_cached_page

```rust
let lookup = self.lookup_cached_page(&key, lsn, ctx).await;
let cached_page_img = match lookup {
    Some((cached_lsn, cached_img)) => {
        match cached_lsn.cmp(&lsn) {
            // there might be WAL between cached_lsn
            // and lsn, we need to check
            Less => Some((cached_lsn, cached_img)),
            // exact LSN match, return the image
            Equal => return Ok(cached_img),
            Greater => unreachable!()
        }
    }
    None => None,
};
```

Speaker notes

If we're lucky, this page is already cached at this LSN. This is backed by a hashmap and fixed size memory buffer of 128MB.

pageserver / GetPage@LSN / reconstruct_value

```rust
let mut reconstruct_state = ValueReconstructState {
    records: Vec::new(),
    img: cached_page_img,
};

self.get_reconstruct_data(key, lsn, &mut reconstruct_state, ctx).await?;
let page = self.reconstruct_value(key, lsn, reconstruct_state).await?;
```

Speaker notes

Either we have a cached page, but not fully up to date, or we have no page at all. either way, we need to reconstruct the new page. Neon's KV page storage is built on top of a custom LSM tree, with 'image' and 'delta' layers. These are either cached in memory, saved to disk, or available in S3. Image layers contain full pages, delta layers contain compressed page diffs

safekeeper / START_WAL_PUSH

```
let (msg_tx, msg_rx) = channel(MSG_QUEUE_SIZE);
let (reply_tx, reply_rx) = channel(REPLY_QUEUE_SIZE);
let handle = Some(WalAcceptor::spawn(tli, msg_rx, reply_tx));

select! {
    // Forward all messages to WalAcceptor
    r = read_network_loop(self.pgb_reader, msg_tx, next_msg).await => r,
    // Forward all messages to Postgres
    r = network_write(pgb, reply_rx, pageserver_feedback_rx).await => r,
};
handle.await;
```

Speaker notes

Postgres now has all the data it needs to perform the write. It updates the page in memory, now we need to save it. This is where the 'safekeepers' come in. Postgres streams out WAL pages to many safekeepers, for... safekeeping...

safekeeper / handle_append_request

```rust
self.wal_store.write_wal(msg.h.begin_lsn, &msg.wal_data).await?;

// flush wal to the disk, if required
if require_flush {
    self.wal_store.flush_wal().await?;
}

// report that the WAL was written
let resp = self.append_response();
Ok(Some(AcceptorProposerMessage::AppendResponse(resp)))
```

Speaker notes

The safekeepers will process all 'append' requests by saving to disk. sending a message back to Postgres that the data was written.

safekeeper / WalBackupTask

```rust
let mut backup_lsn = Lsn(0);

loop {
    self.commit_lsn_watch_rx.changed().await;
    let commit_lsn = *self.commit_lsn_watch_rx.borrow();
    backup_lsn = self.timeline.get_wal_backup_lsn().await;

    backup_lsn_range(
        &self.timeline,
        &mut backup_lsn,
        commit_lsn,
        self.wal_seg_size,
        &self.timeline_dir,
        &self.workspace_dir,
        self.parallel_jobs,
```

Speaker notes

Once data is flushed to several safekeeper disks, one safekeeper will occasionally back up the WAL to S3 or some other durable object store.

pageservers / START_REPLICATION

```rust
let query = format!("START_REPLICATION PHYSICAL {startpoint}");

let copy_stream = replication_client.copy_both_simple(&query).await?;
let mut physical_stream = pin!(ReplicationStream::new(copy_stream));

let mut waldecoder = WalStreamDecoder::new(startpoint, timeline.pg_version);

let mut walingest = WalIngest::new(timeline.as_ref(), startpoint, &ctx).await?;
```

Speaker notes

At this point, postgres is happy that the data has been written, but the work is not yet done. Our safekeepers have saved the WAL, but pageservers do not know about this data. Our pageservers will open a 'physical replication slot' with one of the safekeepers in order to receive the WAL from safekeepers once it is considered commited.

pageservers / START_REPLICATION

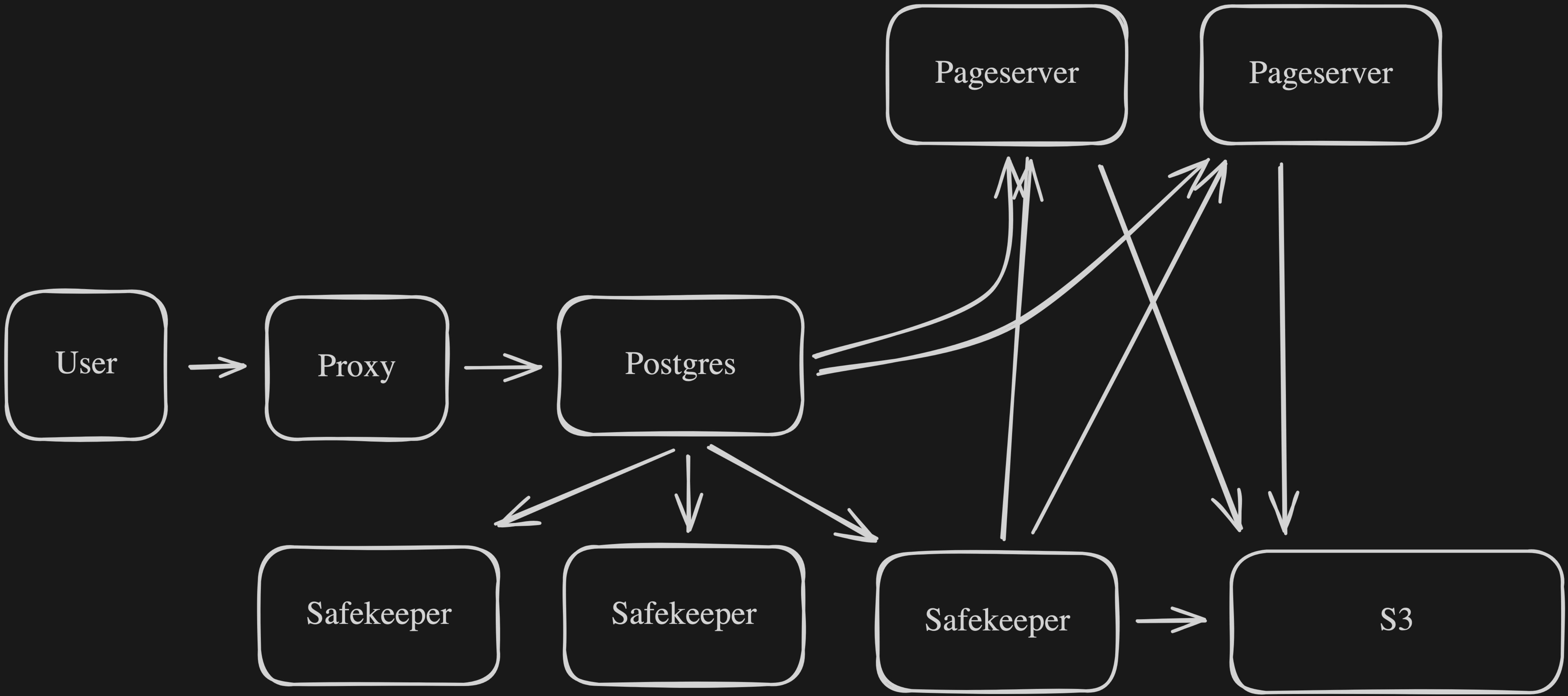
```rust
let data = xlog_data.data();
let startlsn = Lsn::from(xlog_data.wal_start());

waldecoder.feed_bytes(data);

let mut decoded = DecodedWALRecord::default();
let mut modification = timeline.begin_modification(startlsn);
while let Some((lsn, recdata)) = waldecoder.poll_decode()? {
    // Ingest the records.
    walingest.ingest_record(recdata, lsn, &mut modification, &mut decoded, &ctx).await?
}
```

Speaker notes

We read the WAL data from safekeepers, send them to the separate 'walingest' process to rebuild any page data.


Pageserver
Pageserver
User
Proxy
Postgres
Safekeeper
Safekeeper
Safekeeper
S3

Speaker notes

And 10 minutes later, we have built a database in Rust. Easy peasy. The other 120,000 lines of Rust and however much C and Go code is left as an exercise to the reader.

THANK YOU

https://github.com/neondatabase/neon
https://neon.tech/blog/category/engineering

NEON

https://neon.tech
https://conradludgate.com