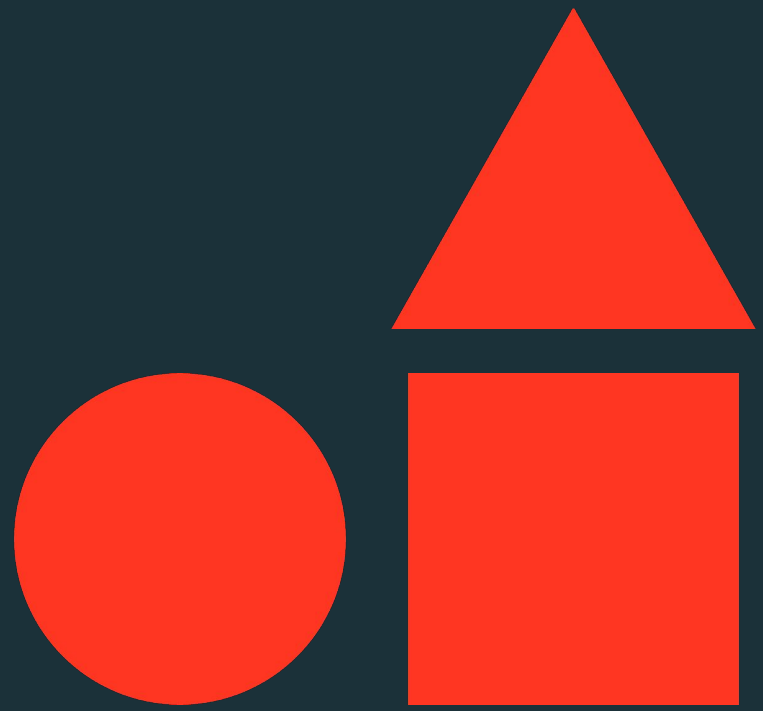
databricks

PostgreSQL as an open data format

100x faster TPC-H queries through direct storage reads

Hristo Stoyanov, Jonathan Katz

May 21, 2026

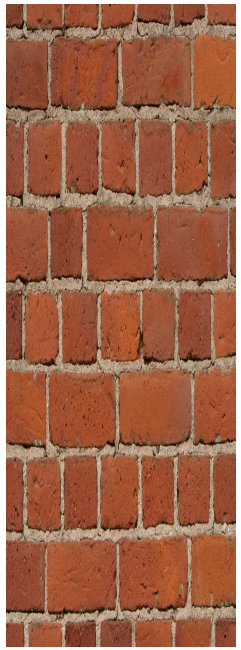


Modern analytics: the lakehouse

OLTP x OLAP: Laws of Physics

OLTP: Online transactional processing

- Get data from a row
- Lots of small reads and writes
- Row-oriented heap, MVCC, index lookups
- Low latency / high throughput



OLAP: Online analytical processing

- Analyze data in a few columns
- Very large reads, batched writes
- Columnar layout, vectorized, MPP
- "Go as fast as possible over gigantic amounts of data as cheaply as possible"

It's very hard for both to co-exist

Faster Horses

Speeding up existing technology

- Massively parallel processing (MPP)
- Vectorized execution
- JIT-compiled plans
- Optimizers + statistics hyperfocused on OLAP

Reinventing Wheels

Rediscovering existing RDBMS features

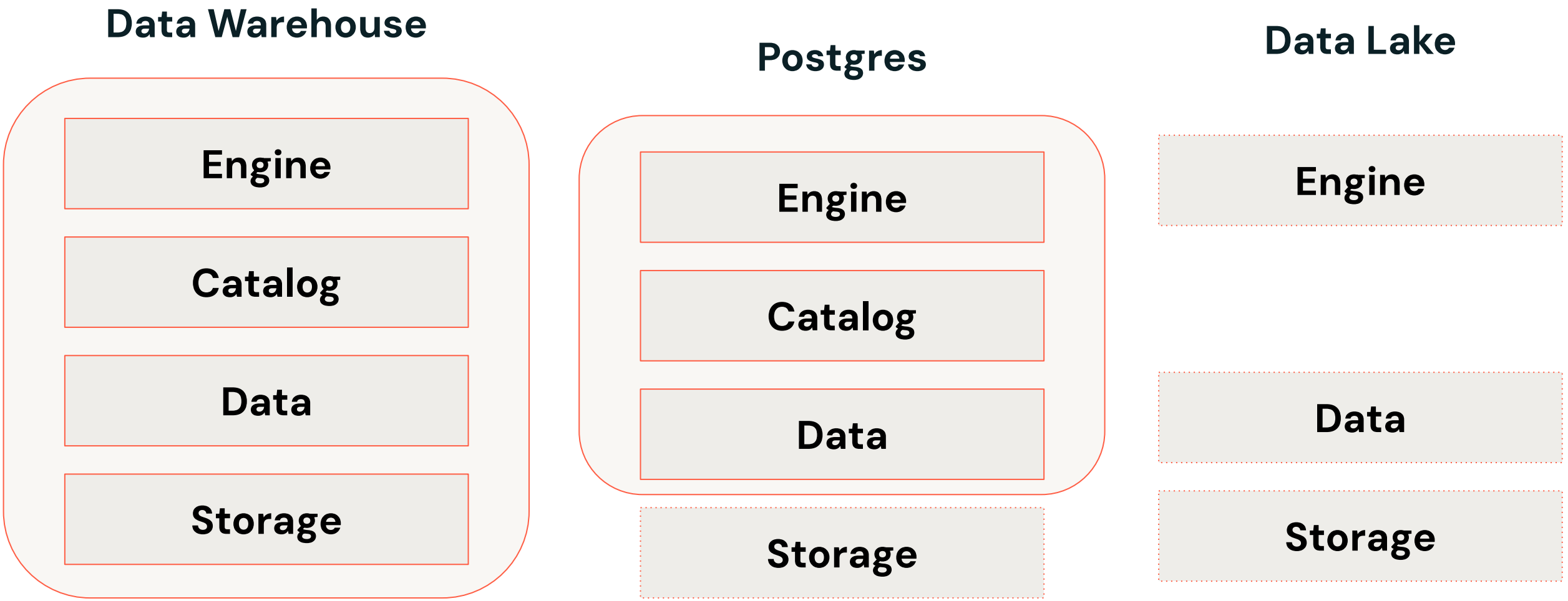
- Modern analytical systems rebuilt features Postgres had decades ago
  - MVCC
  - Access control
  - Multi-statement SQL transactions
- ...but built on top of cloud object storage

New Realities

Innovations that changed how we do analytics

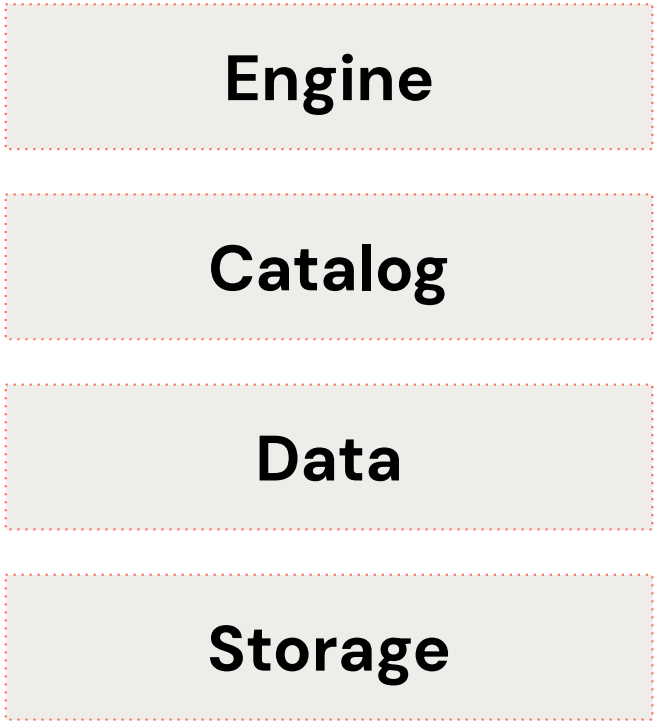
- **Hardware**
  - Specialized disk arrays gave way to cloud object storage (cheap, elastic, durable)
  - Compute is ephemeral and scales to the workload instead of peak
- **Data**
  - Open columnar formats: Parquet, Arrow
  - Open commit log: Delta, Iceberg
- **Provisioning Model**
  - Multiple computes can read and write(!) to shared storage

Data warehouses and data lakes



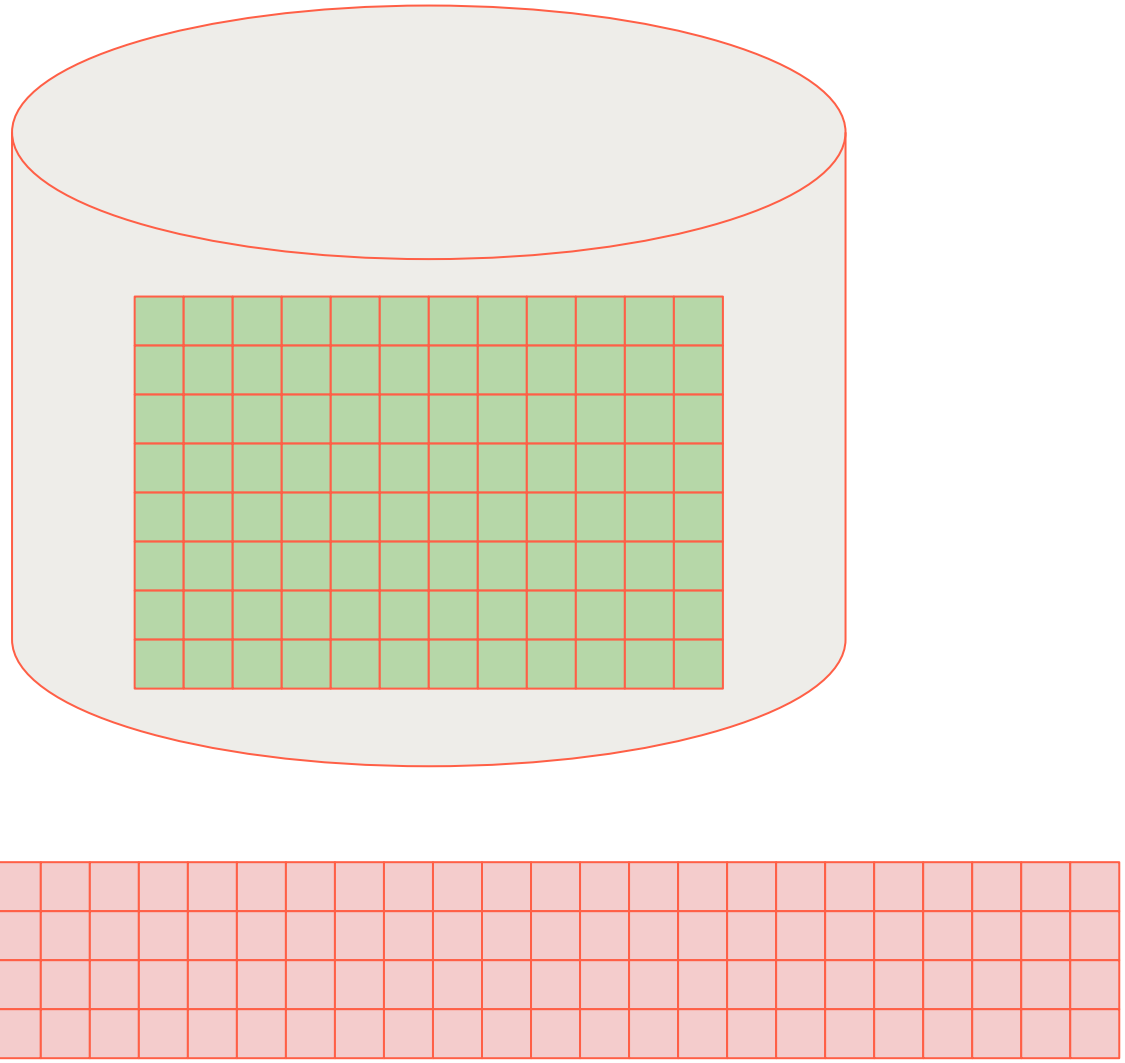
Enter the “lakehouse”

- **Unified** - Single architecture that allows governance of all data components
- **Open** - Open source at all levels (data, catalog, engine)
- **Scalable** - Consumption based that scales with workload

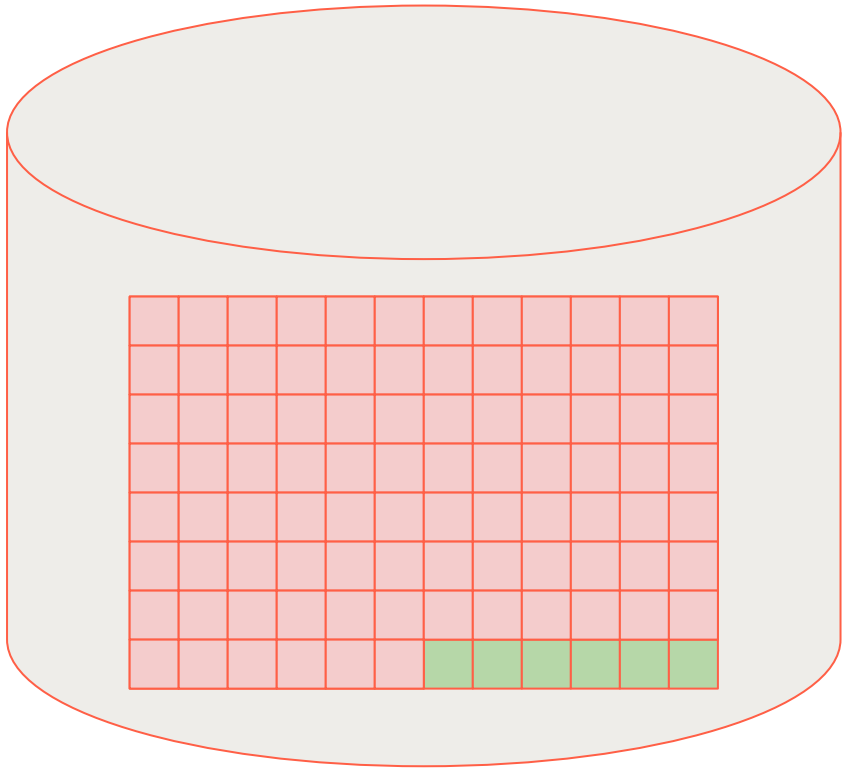


Result - Shared Storage. Multiple engines can safely read/write to the same storage; no lock-in on any component

What about PostgreSQL?



```
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table JOIN table2 ON
table2.ref_id = table.id WHERE
table.id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table2 WHERE
insert_date = CURRENT_DATE AND
table2.ref_id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
```



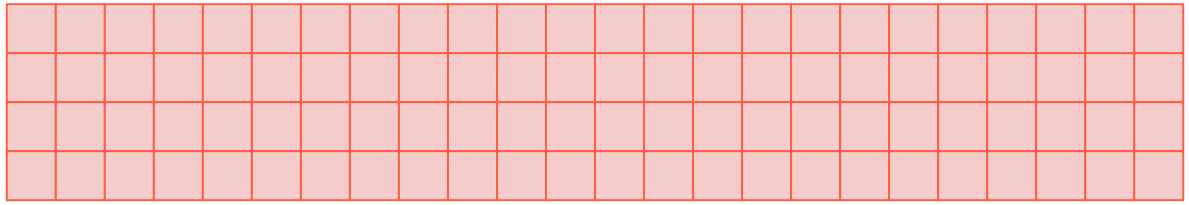
```
SELECT
  insert_date,
  count(*)
FROM table
GROUP BY insert_date
ORDER BY insert_date;
```

```
SELECT * FROM table
WHERE id = $1
```

```
SELECT * FROM table
WHERE id = $1
```

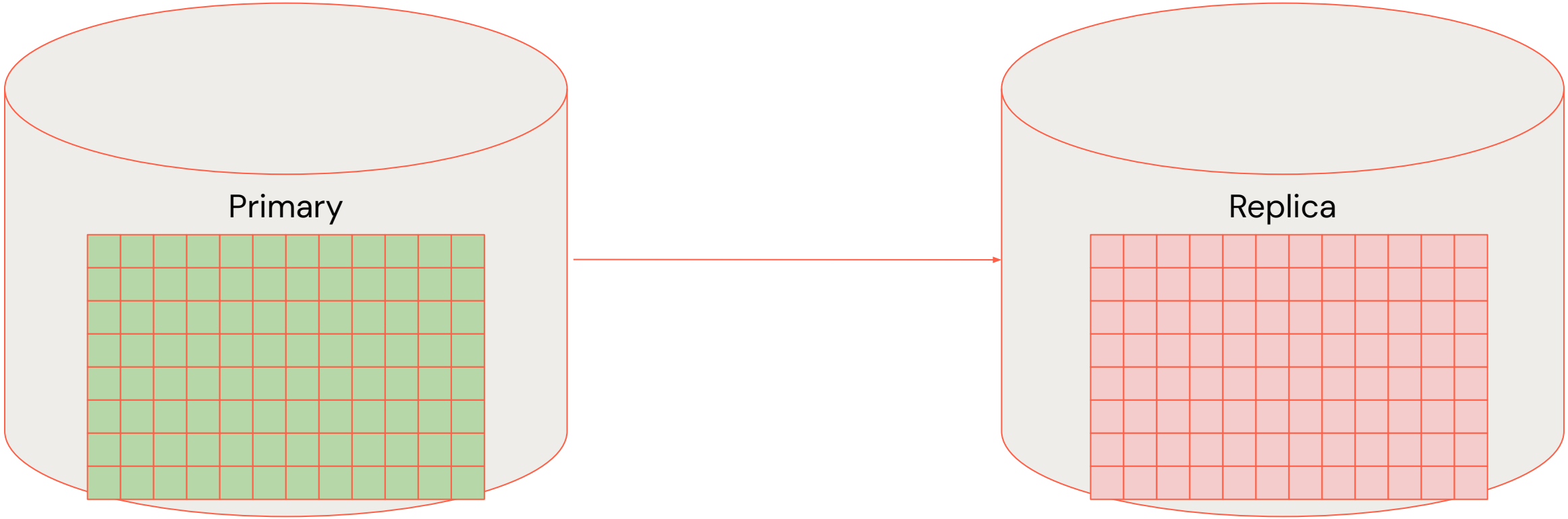
```
SELECT * FROM table
WHERE id = $1
```

```
SELECT * FROM table
WHERE id = $1
```



```
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
SELECT * FROM table WHERE id = $1
```

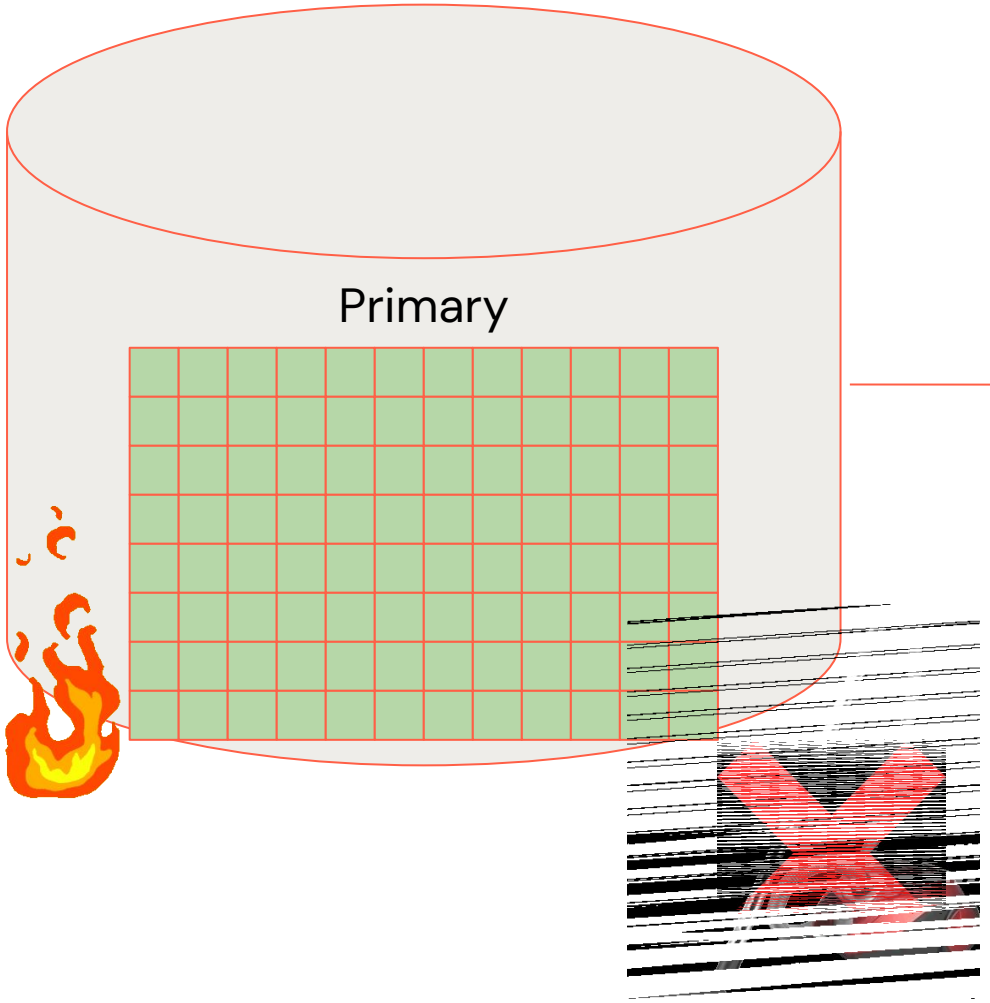
```
SELECT
  insert_date,
  count(*)
FROM table
GROUP BY insert_date;
```



```
DELETE FROM table
WHERE insert_date < CURRENT_DATE -
90;

VACUUM table;
```

```
SELECT
  insert_date,
  count(*)
FROM table
GROUP BY insert_date;
```



How can you modern OLAP with PostgreSQL?

Columnar in-process

- Citus columnar, Hydra, pg_mooncake
- Postgres storage; Postgres execution
- Hard to make columnar amenable to MVCC point writes

Embedded OLAP engine

- pg_duckdb, pg_lake
- Read Iceberg/Parquet data into Postgres resultsets
- Competes with OLTP resources

Move data out

- Logical replication
- Eventually consistent
- Schema drift, type fidelity
- Replication slot pain

What do we sacrifice when we move data?

- **MVCC consistency** between OLTP and OLAP views
- **Catalog Integration:** PostgreSQL types, permissions, and schema serve as the single source of truth
- **Tradeoff between compute isolation / freshness**
  - In-process embedded solution
  - Replicate data out

We keep choosing between freshness, isolation, and transactional truth

Analytics directly on OLTP data

Making the new reality

Q1: How to access Postgres data?

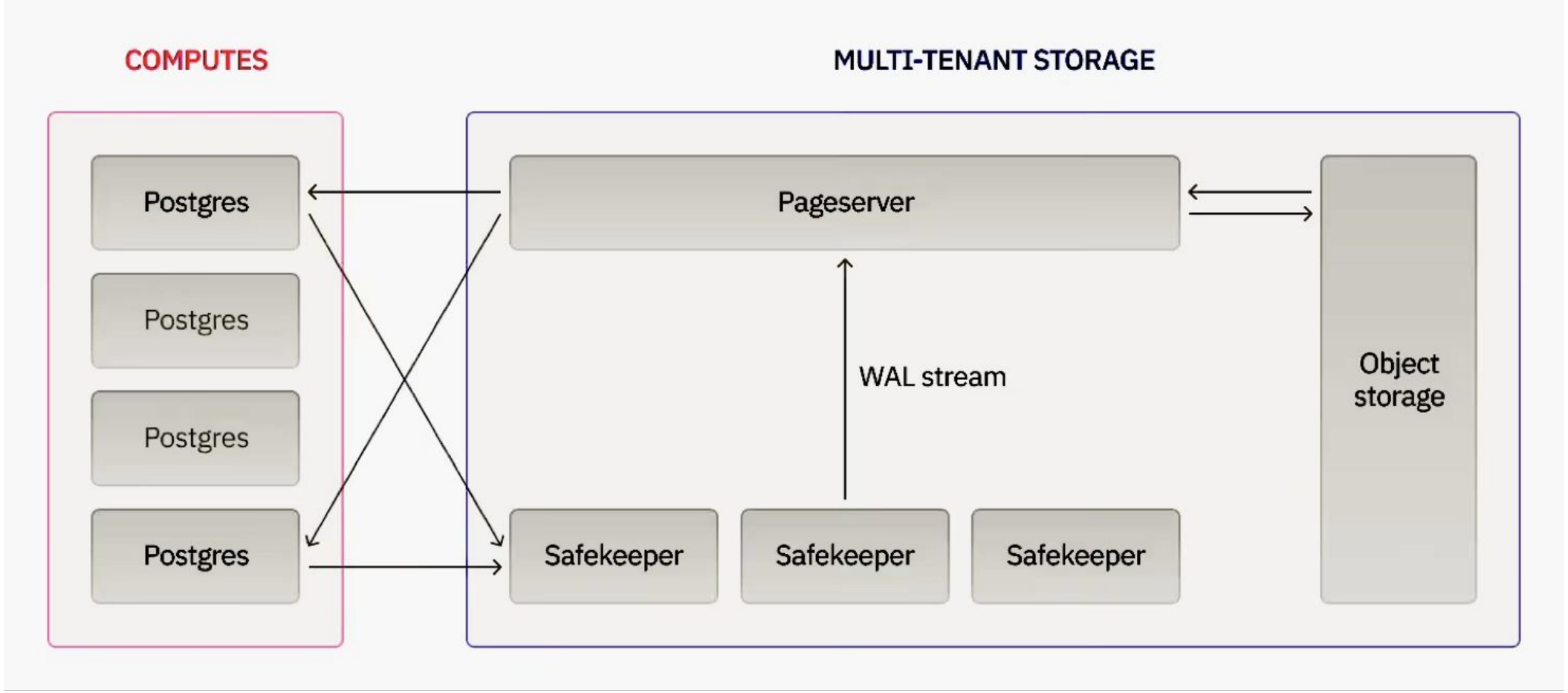
Making the new reality

Q2: How to interpret Postgres data?

Isn’t PostgreSQL storage “open”?

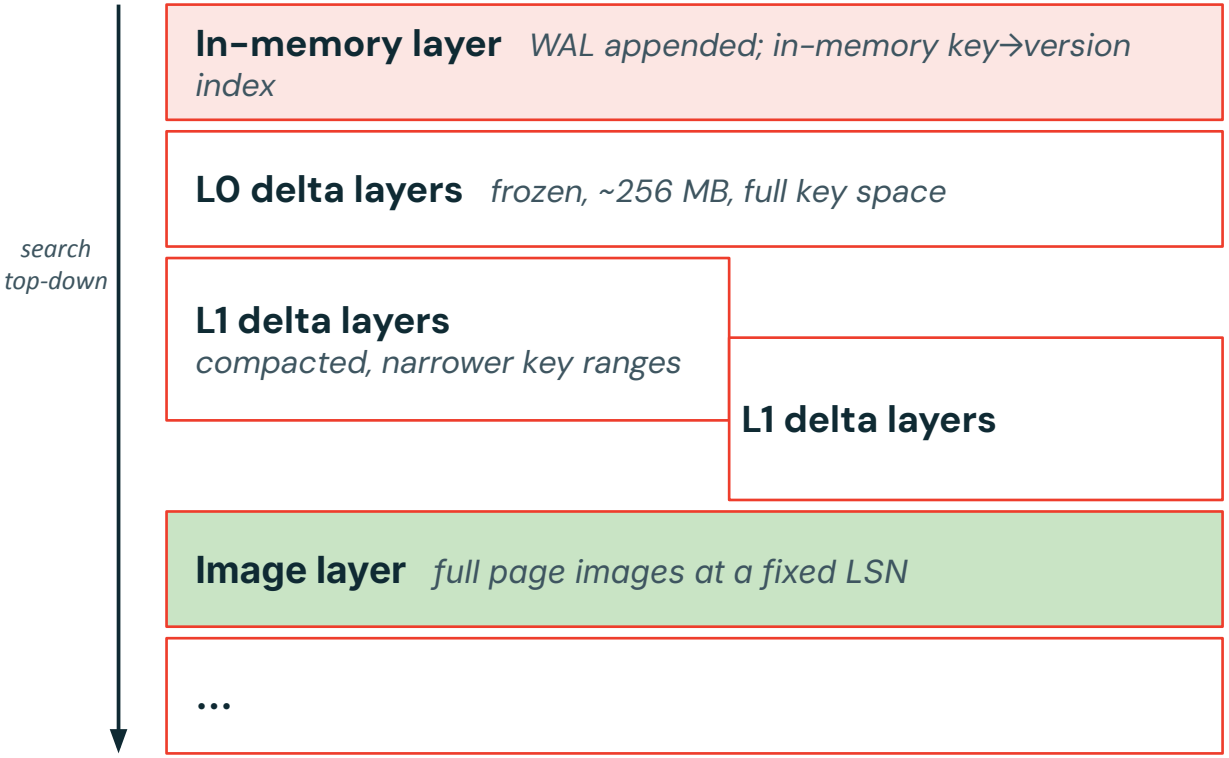
- Postgres already has a well-defined on-disk format at a specific version
- Transactional semantics are available in a reference implementation
- What if we treated relation files as an open format and read them?
  - Ignore permissions
  - custom types,
  - Extensions,
  - indexes for a bit left as an exercise for the reader
- How hard is it?

Neon storage architecture



Neon storage format

Stacks of immutable layers

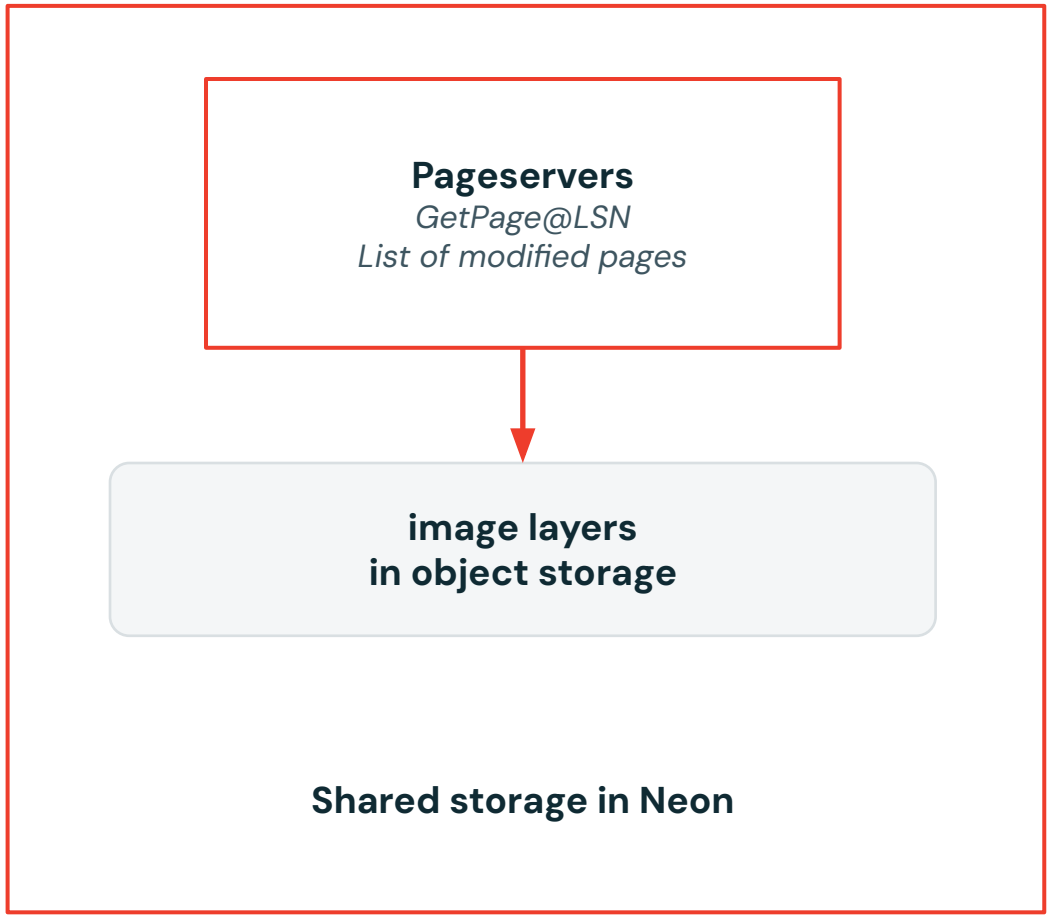


- Layers
  - Cover key range × LSN range
  - Contain full-page images & WAL records
  - Immutable
  - Uploaded to blob storage
- GetPage@LSN
  - Reads walk newest → oldest until a full page image plus any WAL entries can reconstruct the page as of the request LSN
- Base backup
  - Checkpoint file
  - SLRU segments including CLOG & multixact
- **Outcome**: elastic storage with ephemeral computes

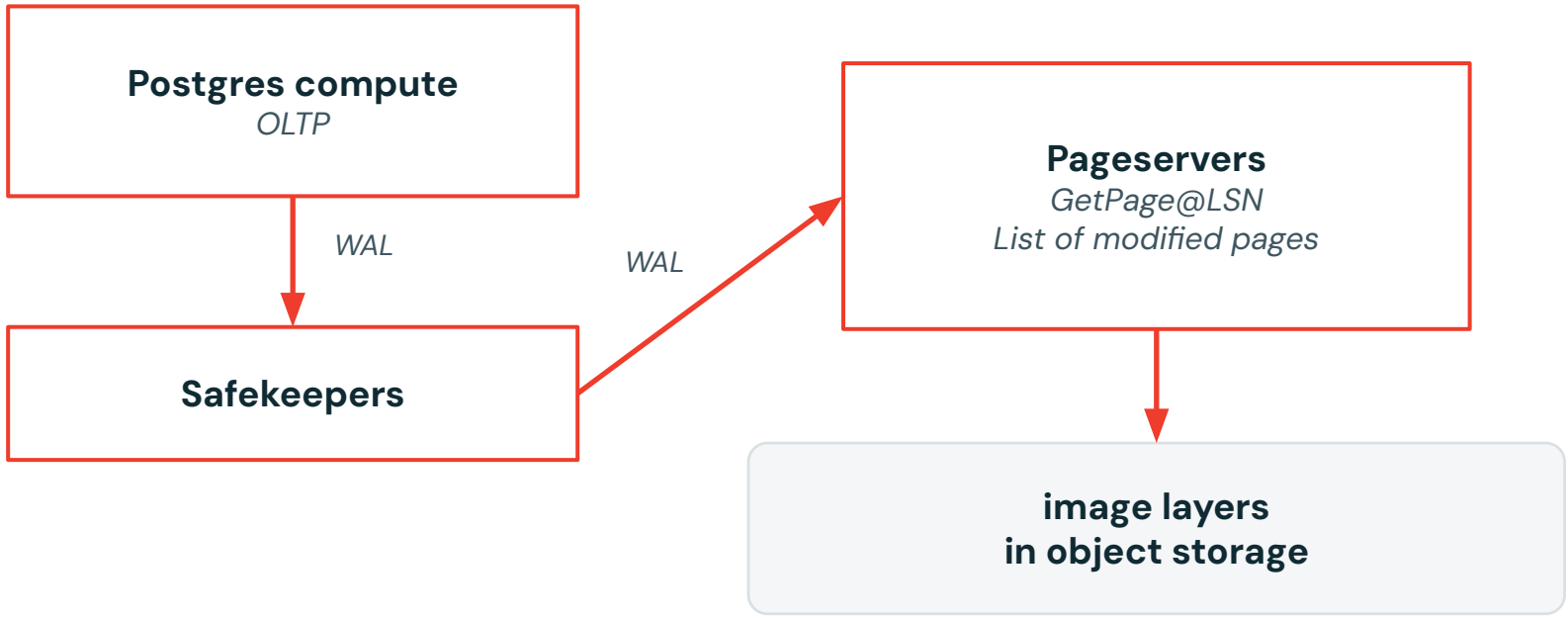
OLTP data on shared storage now…

Can you read it?

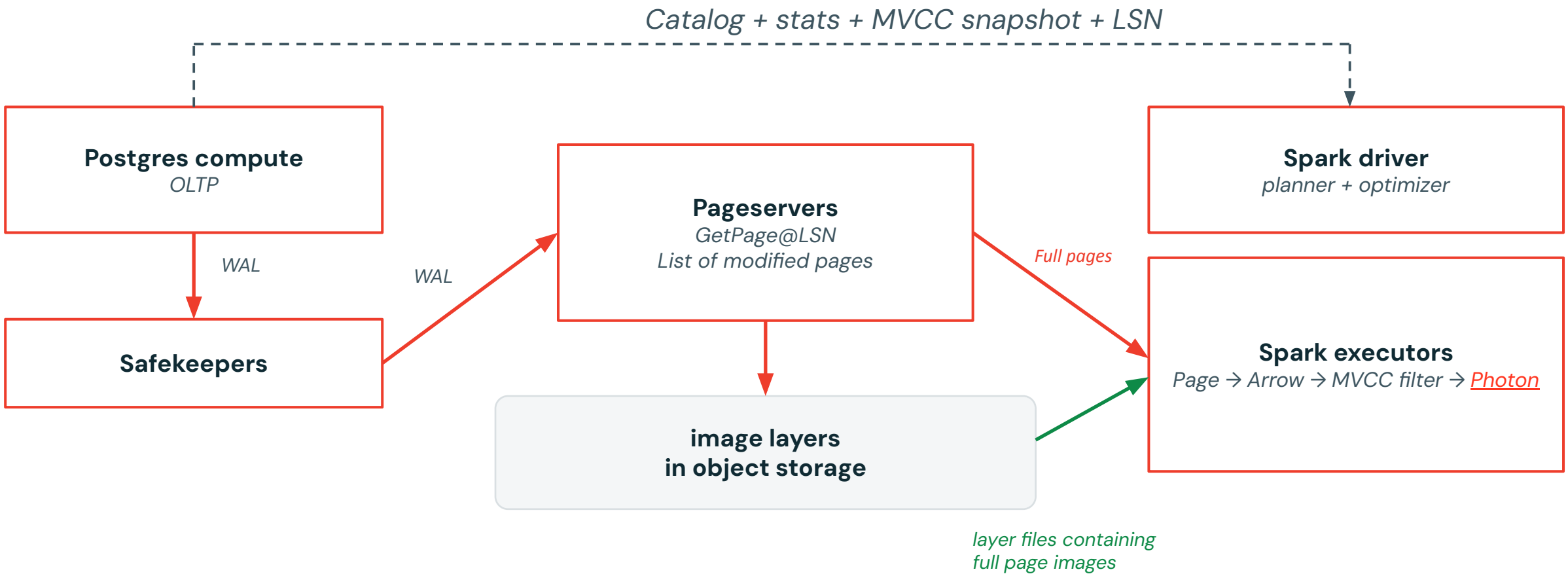
End-to-end architecture



End-to-end architecture



End-to-end architecture



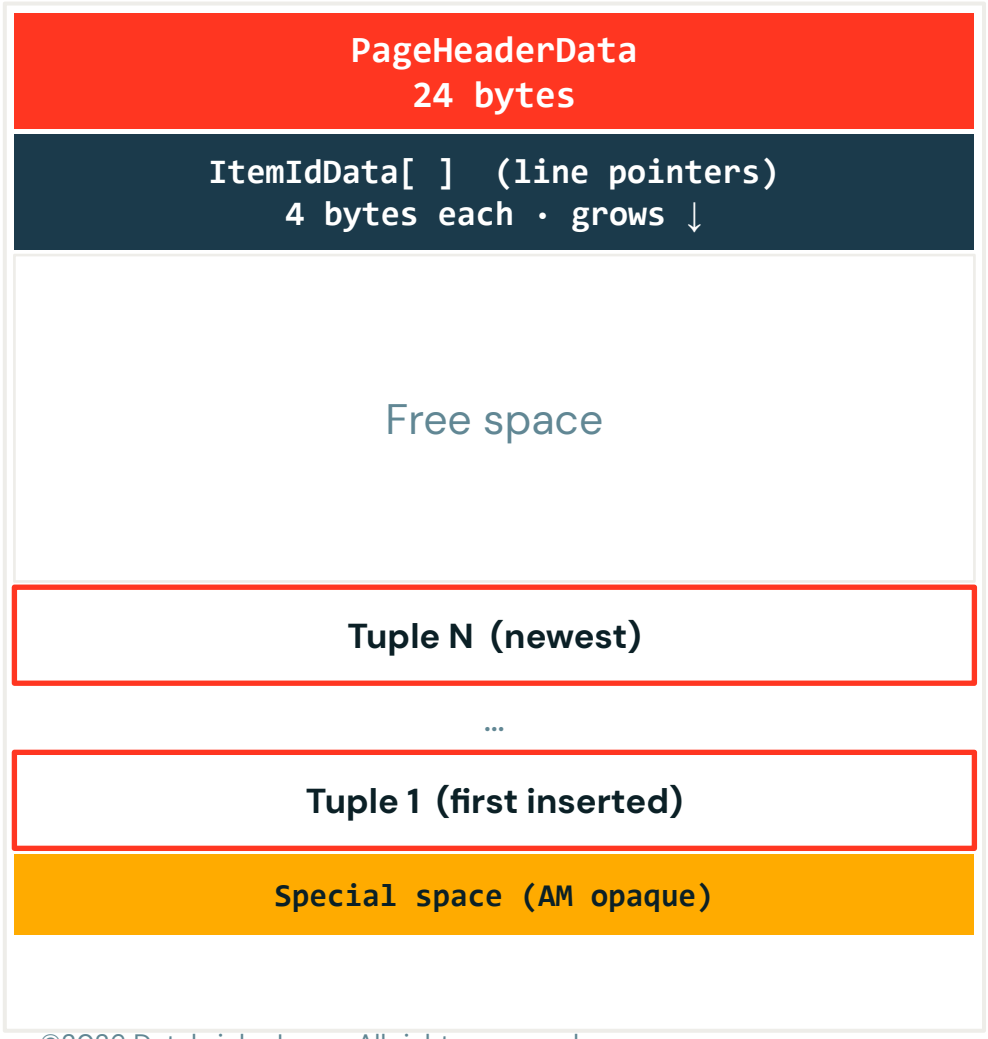
Pageserver provides storage & SLRU

- Pageserver embeds Postgres and runs WAL redo to perform pages
- Correctly resolving MVCC requires
  - Table data
  - CLOG segments, calculated by pageserver in Neon/Lakebase
  - Checkpoint file with range of valid xids & multi-xids
  - Multixact segments
- GetRelSize @ LSN
- GetSlruSegment @ LSN
- GetPage @ LSN

Making the new reality

Q2: How to interpret Postgres data?

Heap page format



0x0000

0x0018

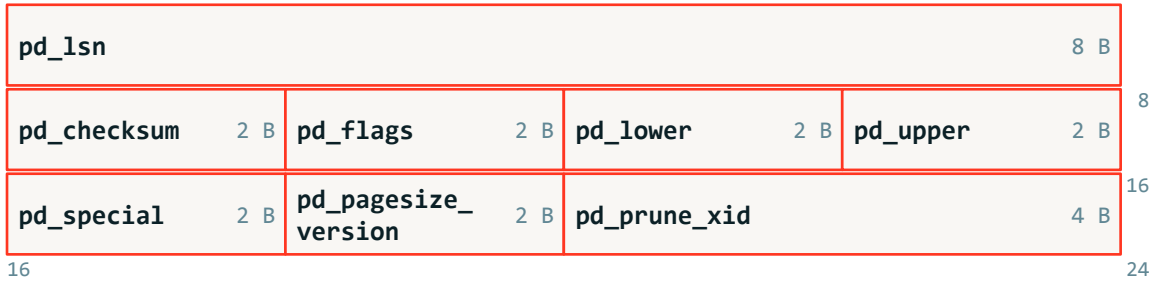
pd_lower

pd_upper

tuples grow ↑

pd_special
0x2000 (8192)

PageHeaderData: 24 bytes



ItemIdData — 32 bits (4 bytes)

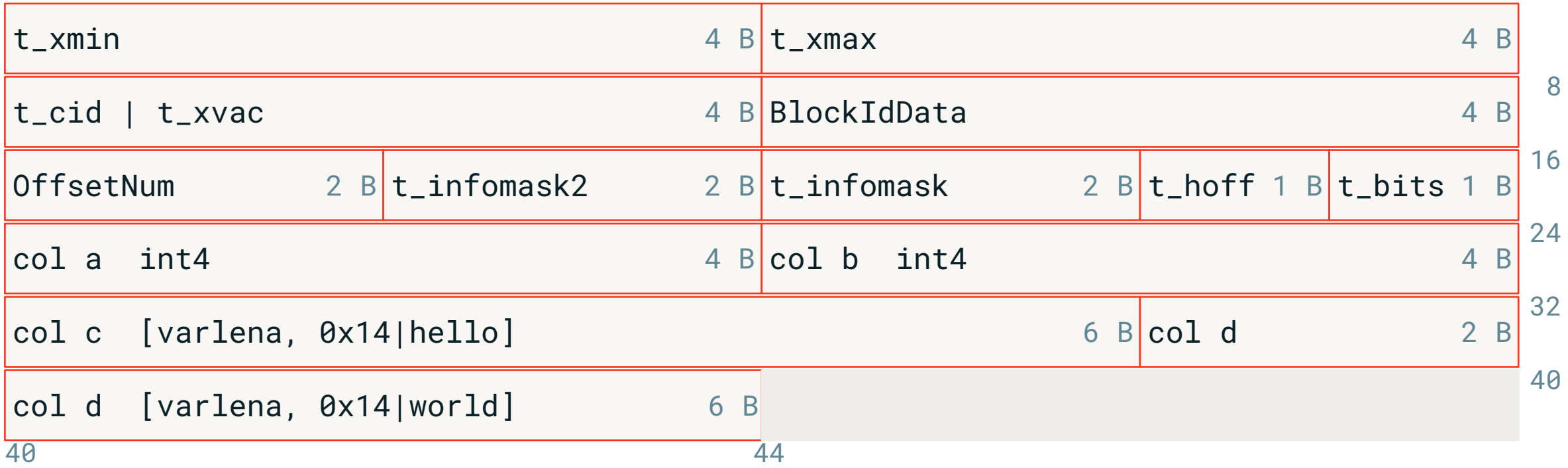
0 = LP_UNUSED 1 = LP_NORMAL 2 = LP_REDIRECT 3 = LP_DEAD

PostgreSQL format: tuple

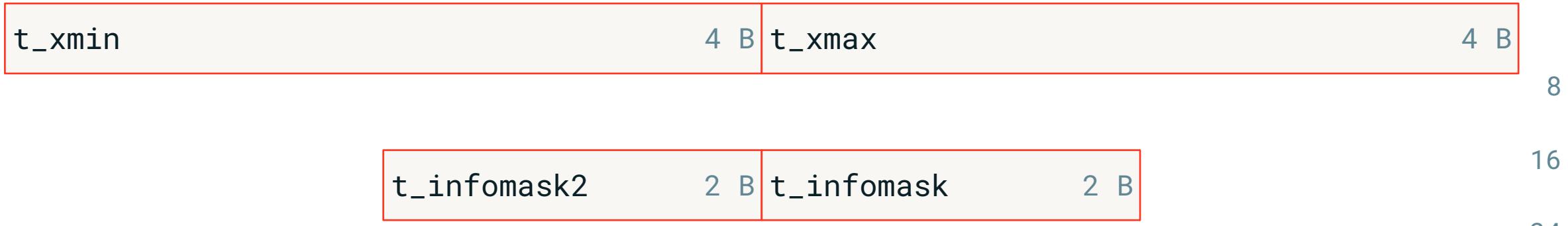
- Tuple, things you need to process to interpret data externally
  - Headers, simpler
  - Visibility, hard
    - frozen tuples, clog, multixact
  - TOAST, hard
    - Index navigation or run a join with reltoast file
- Row orient vs PAX
  - PAX: fixed-size page (e.g. 8KiB) but partition attributes across

PostgreSQL format: tuple

```
create table t (a int4, b int4, c text, d text);
```



PostgreSQL format: tuple



- Compare XIDs with xip list, check CLOG & multixact fields

Catalog metadata required to run a query

- Fetch tables, columns, types at txn snapshot + LSN
  - Export snapshot, LSN
- Catalog information required to interpret
  - List of attributes with types, nullability, attisgenerated/attisdropped
  - Statistics
- Still need to connect to PG
  - Potentially extract types from pg_catalog in the future
  - Gift of incredible flexibility of type system(!) but also very hard

Custom types complexity

- Custom types require code in many different varieties
  - C ABI-compatible can be invoked from non-PG engines too (theoretically)
  - PL/pgSQL only ever runs on PG, no other VM or compiler exists
  - Major constraint one has to operate within
- There is no way to know what a custom type is without external operations information

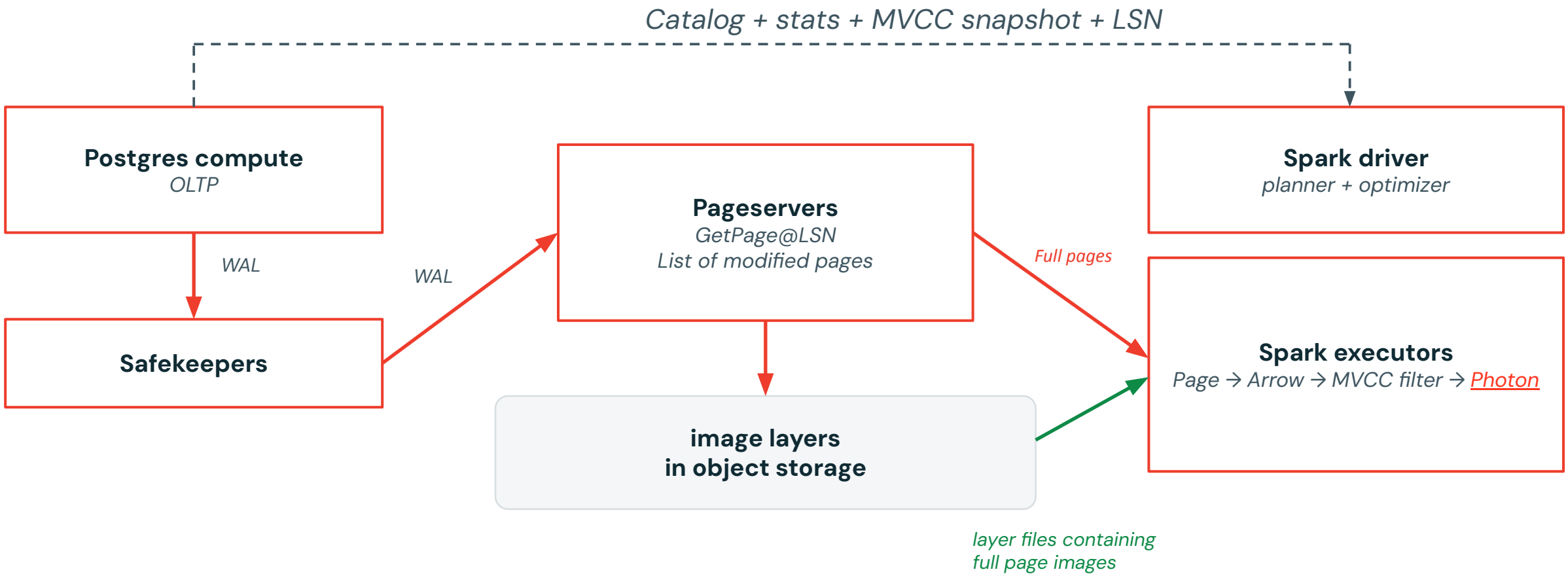
Transactional semantics

- Return consistent results with PG
  - Export txn snapshot
  - + catalog reads @ txn snapshot
  - + LSN
  - + clog/multixact resolution
  - Aim for consistency with PG primary

How scans work end-to-end

- DBSQL driver reads metadata from Postgres
  - Table stats: reltuples, distinct-value counts
  - Transaction snapshot
  - Catalog information @ txn snapshot
- Assemble relation file from two sources
  - Layers from S3
  - Recent pages from the pageserver gRPC API
- Applies MVCC
  - Export a snapshot from a regular Postgres transaction:
    - xmin, xmax, xip list
  - SLRU segments are read from pageserver
  - Visibility rules reimplemented inside the analytics engine

End-to-end architecture



Results

Does it work?

Demo

Open Problems

Where can we go from here?

Open problems

- **Reconstructing Postgres transactional semantics outside the server**
  - Visibility, type decoding, TOAST resolution, and SLRU access live inside Postgres process
  - Other engines have to reimplement heap visibility
- **Bandwidth path through the pageserver**
  - Recent pages still go through the pageserver gRPC
  - Minimize metadata reads from Postgres primary
  - How to read data & *metadata* directly from shared storage
- **Custom + complex types**
  - Composite, enums, range types have well-defined behavior
  - Arrays hard, potentially feasible
  - Base types practically impossible?
- Not looking to align query semantics (other analytics engines don't do this)

Next Steps

- Q1: Should we build an interface to get a consistent snapshot of a relational file from shared storage?
  - API to perform reads of nonrelational metadata (CLOG, MultiXact)
  - Ability to read relational files at any LSN from storage

- Q2: What should the file format for Postgres relational data (pages, forks, TOAST files, etc.) containing multiple tables across time be?