# Local-first apps using Postgres logical replication

## Building a high-scalability sync engine with dynamic partial replication

**Conrad Hofmeyr**
*Co-Founder & CEO, JourneyApps*
Postgres Conference 2024
April 19, 2024

# Local-first apps using Postgres logical replication

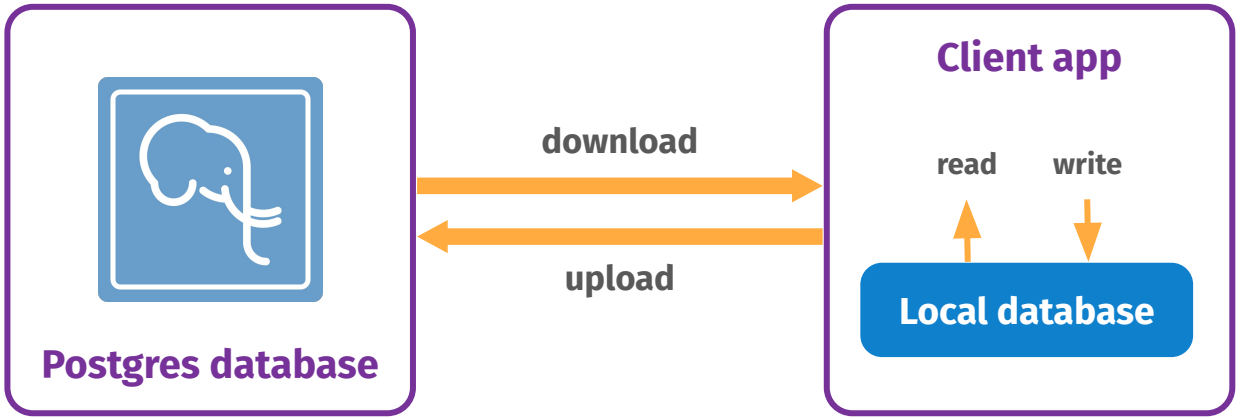
*Building a high-scalability sync engine with dynamic partial replication*

1. What is local-first? Why local-first?
2. Why we focused on Postgres & SQLite
3. System design, architecture & implementation
4. Postgres logical replication challenges & solutions

# What is local-first?

Local-first (some overlap with “offline-first”) is an architecture where app code works directly with a **client-side embedded database** which **automatically syncs with a backend database** in the background.

Reads and writes go to the **local database first.**



# Why local-first app architecture?

## For app developers

Simple **state management**

**Simplified backend**: Reduced API development burden

Lower **backend compute load**, dependency, and cost

## For end-users

Instantly **reactive UX**

Apps are always available: **Work offline**

Real-time multi-user **collaboration**

# Why we focused on Postgres


Full-stack platform for offline-first industrial apps

Spin-off stand-alone sync engine

# Why we focused on Postgres

| Backend Database | Database Type | Sync Layer | Client Database |
| --- | --- | --- | --- |
|  | NoSQL | Atlas Device Sync |  |
|  | NoSQL | PouchDB.replicate() |  |
|  | SQL | ? |  |

# Choosing Postgres & SQLite



- SQL as a language remains **more universal** and well-known than NoSQL syntaxes.
- SQL is very effective for **advanced queries** e.g. aggregations.
- **SQLite*** is hard to beat on the client-side:
    - **Performance, flexibility & advanced functionality** (aggregations, joins, advanced indexing, JSON)
    - **Battle-tested** (a trillion SQLite databases are in active use)

** Our architecture allows swapping SQLite with other client-side databases.*

# Design, architecture & implementation

PowerSync Postgres<>SQLite sync engine

## System design, architecture & implementation

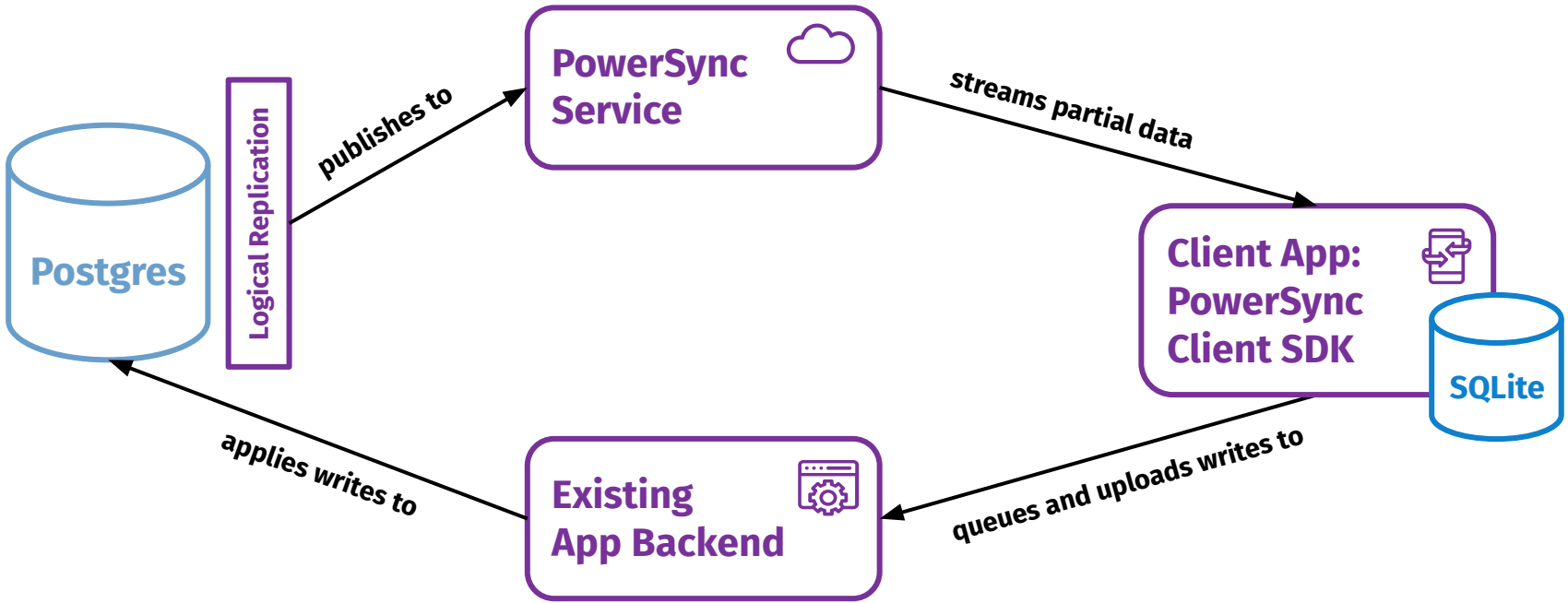
a. **Design objectives**
b. **High-level architecture**
c. Implementing *dynamic partial replication*
d. Syncing local writes back to Postgres
e. Server-authoritative architecture: No CRDTs needed
f. Guaranteeing consistency
g. Guaranteeing data integrity

# Design objectives

1. **Treat the Postgres database as sacrosanct:**
    - Work with standard Postgres databases, non-invasively.
      Require minimal-to-no changes to schema, configuration, etc.
    - Don’t bypass the developer’s existing business logic, authorization and validation.
      Don’t write directly to their Postgres database.
2. **Dynamic partial replication.**
    - Flexibly sync only the needed data to each user.
3. **Low maintenance**.
    - For example: schema and data migrations.
4. **Strong consistency guarantees**.

# High-level architecture

Read/download path vs. write/upload path:



## System design, architecture & implementation

a. Design objectives
b. High-level architecture
c. **Implementing *dynamic partial replication***
d. Syncing local writes back to Postgres
e. Server-authoritative architecture: No CRDTs needed
f. Guaranteeing consistency
g. Guaranteeing data integrity

# Implementing *dynamic partial replication*

**Objective:** Get relevant subsets of data to users.

**Naive approach:** Allow clients to make “arbitrary” database queries and attempt to keep their result sets in sync.

- However: Any client can go offline and come back online at any later time:
  - Would need to sync the delta for their specific queries, relative to their specific current sync state.

*Problematic for high-scalability:*

  - *Large database, high write volume, large number of concurrent users.*

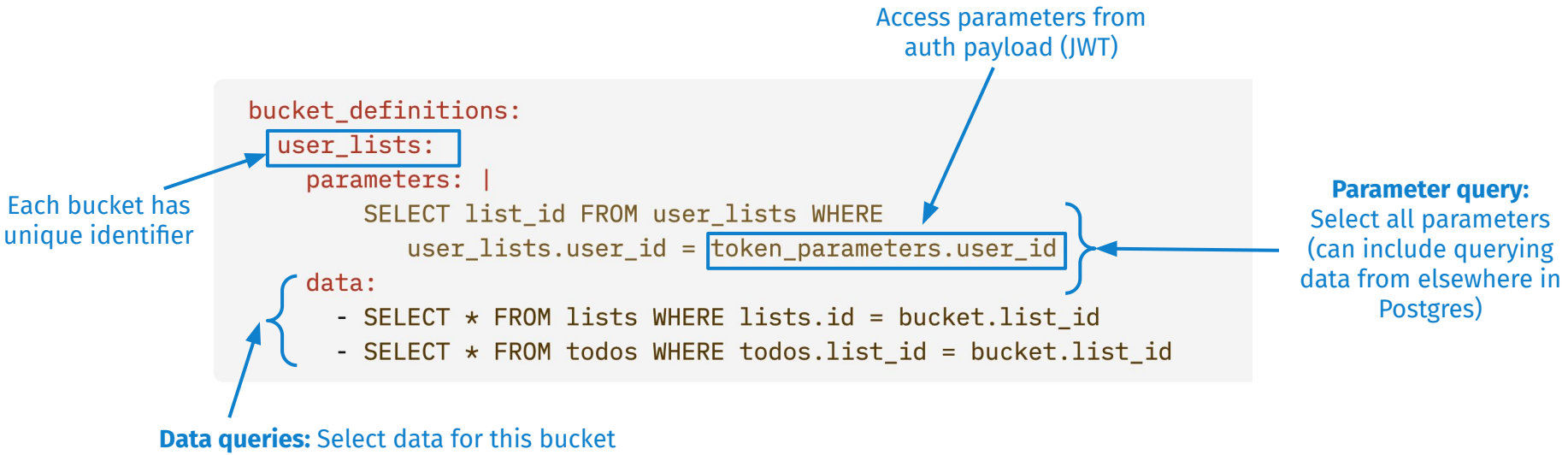
# Implementing *dynamic partial replication*

**Our chosen design:**

- Allow for pre-defining **“buckets” of data**
- Buckets of data can be **shared between users** where applicable.
- **Efficiently track changes** to data in the buckets

# Dynamic partial replication: How it works

**1. Developer defines declarative 'sync rules' with SQL syntax, grouping data into buckets.** Buckets can reference parameters from the client.

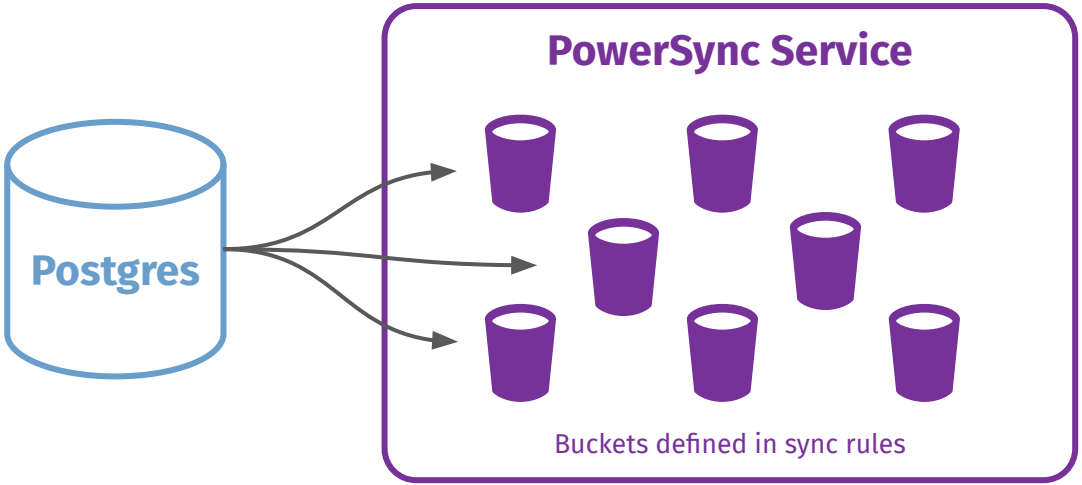


# Dynamic partial replication: How it works

## 2. Based on sync rules, replicate and pre-process data from Postgres:

Replication initially performed by taking a snapshot of tables defined in the Sync Rules (with **regular SQL queries**), then incrementally (with **logical replication**)

- *Checkpoints* keep track of Postgres **LSNs** in replication stream (more in this later)

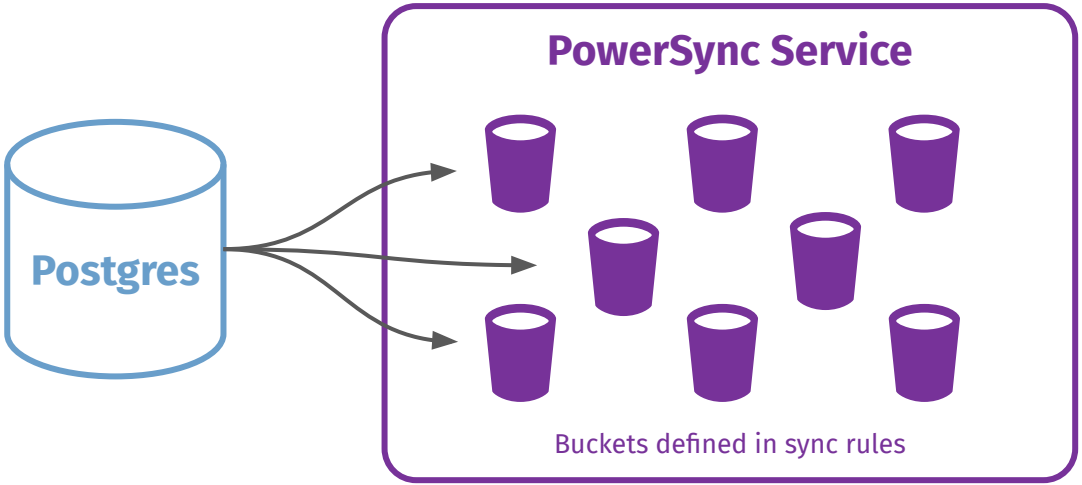


# Dynamic partial replication: How it works

## 2. Based on sync rules, replicate and pre-process data from Postgres:

Pluggable storage is used in the PowerSync Service to store **operation history.**

- Effectively a rebuildable persistent cache outside of Postgres, keeping it clean.



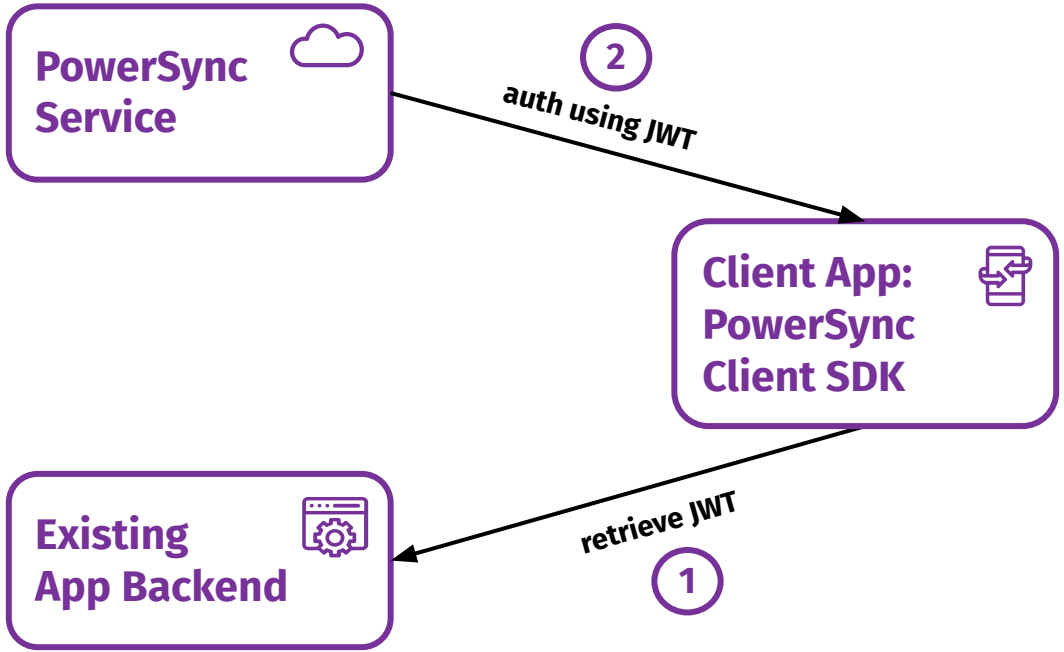
# Dynamic partial replication: How it works

## 2. Based on sync rules, replicate and pre-process data from Postgres:

- The **recent history of operations** on each row is stored.
  - Operation history is compacted for performance.
- **Rows are represented as JSON.**
- **Operations indexed by “operation ID”** (`op_id`), a strictly incrementing integer ID
  - Allows ordered list of operations to be queried efficiently

# Dynamic partial replication: How it works

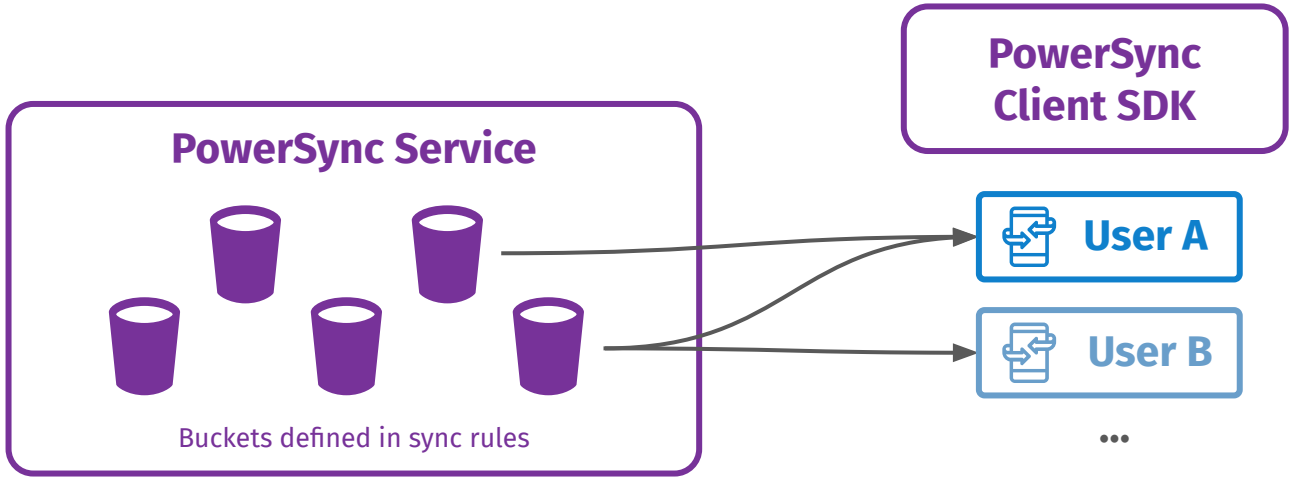
### 3. Authenticate users using JWTs.



# Dynamic partial replication: How it works

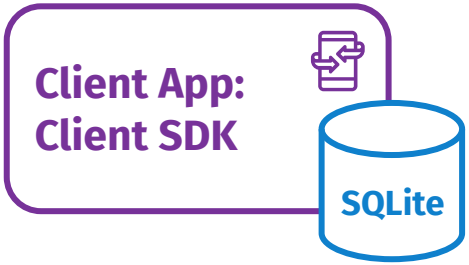
## 4. Streaming sync of bucket data from PowerSync Service to clients:

For buckets that apply to the user: Continuously stream any operations added.



# Dynamic partial replication: How it works

## 5. On the client, data is persisted to SQLite




- **Type mapping** from Postgres → SQLite
- Replicated data is **stored in schemaless format** in SQLite.
- The **application defines a client-side schema** with tables, columns and indices.
- The schema is applied as **SQLite views** on top of the schemaless data.
- **Live reactive query hooks:** Update UI when data changes.

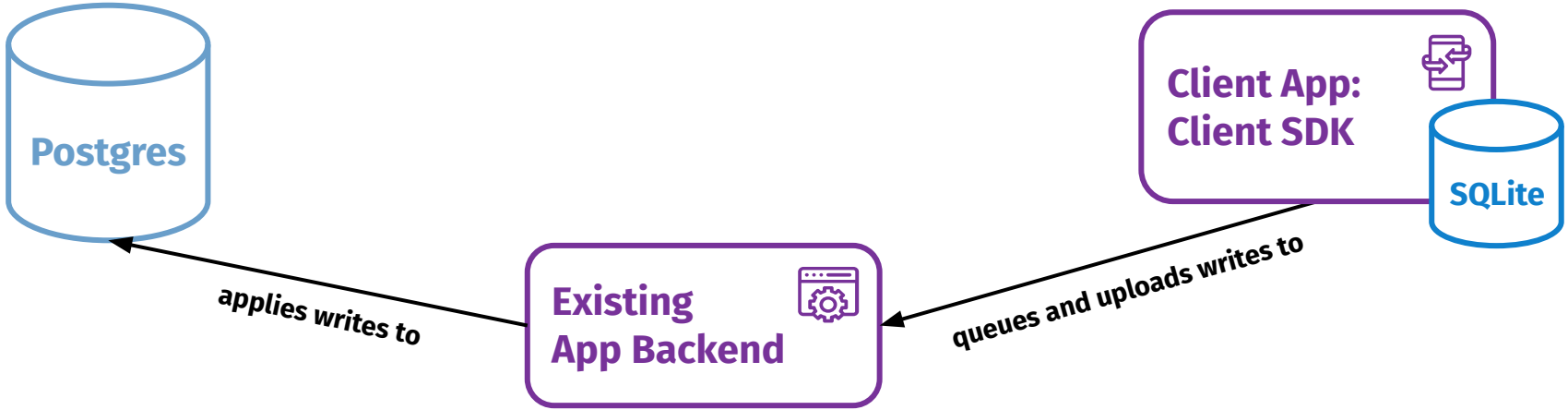
```
export const AppSchema = new Schema([
  new Table({
    name: 'todos',
    columns: [
      new Column({ name: 'list_id', type: ColumnType.TEXT }),
      new Column({ name: 'created_at', type: ColumnType.TEXT }),
      new Column({ name: 'completed_at', type: ColumnType.TEXT }),
      new Column({ name: 'description', type: ColumnType.TEXT }),
      new Column({ name: 'completed', type: ColumnType.INTEGER }),
      new Column({ name: 'created_by', type: ColumnType.TEXT }),
      new Column({ name: 'completed_by', type: ColumnType.TEXT })
    ],
    indexes: [new Index({ name: 'list', columns: [new IndexedColumn({ name:
  }),
```

## System design, architecture & implementation

a. Design objectives
b. High-level architecture
c. Implementing *dynamic partial replication*
**d. Syncing local writes back to Postgres**
e. Server-authoritative architecture: No CRDTs needed
f. Guaranteeing consistency
g. Guaranteeing data integrity

# Writes to SQLite: Syncing back to Postgres

- Writes to the local SQLite database also placed in an **upload queue.**
- Developer **defines their own function** for uploading these changes to their backend, and from there, writing to Postgres. (Client SDK handles retries.)
    - Allows existing business logic, fine-grained authorization, validations and server-side integrations to be honored.



## System design, architecture & implementation

a. Design objectives
b. High-level architecture
c. Implementing *dynamic partial replication*
d. Syncing local writes back to Postgres
**e. Server-authoritative architecture: No CRDTs needed**
f. Guaranteeing consistency
g. Guaranteeing data integrity

# Server-authoritative: No CRDTs needed

- **CRDTs are special data structures where changes can be merged in any order**, and each replica converges to the same state.
- Alternative to CRDTs: **always merge changes in the same order**.
- Our system uses an architecture where there is an authoritative **source of truth regarding the global order of operations**
    - Derived from the Postgres logical replication stream.
- Therefore, our system **does not need CRDTs** *on a protocol level.*

# Server-authoritative: No CRDTs needed

**Provides simplicity and customizability for the developer:**

- The developer's **backend can reject and accept changes** as needed, and clients converge to the server's authoritative state.
- The developer's backend can apply **custom conflict resolution** (including using CRDT data structures stored as blob data in Postgres)

## System design, architecture & implementation

a. Design objectives
b. High-level architecture
c. Implementing *dynamic partial replication*
d. Syncing local writes back to Postgres
e. Server-authoritative architecture: No CRDTs needed
**f. Guaranteeing consistency**
g. Guaranteeing data integrity

# Guaranteeing consistency

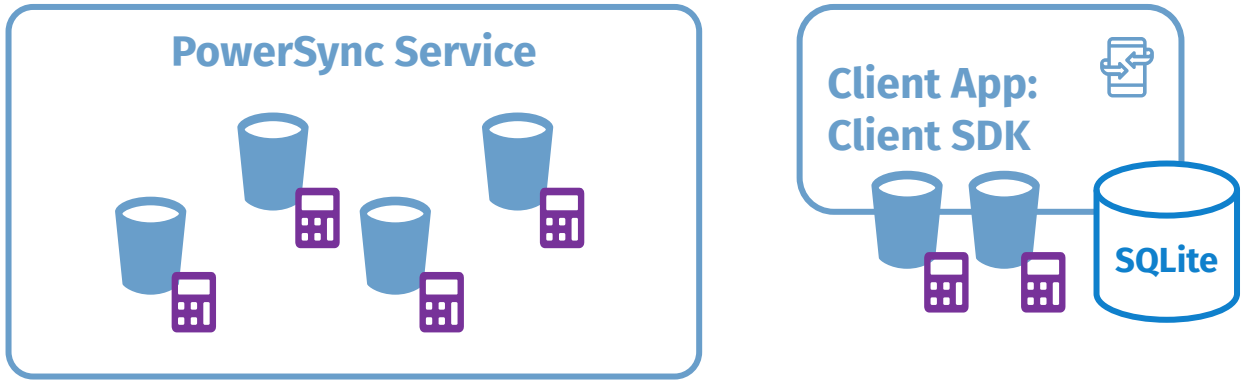
- We keep track of **write checkpoints** on the server.
  - Checkpoints have an associated "**operation ID**" (`op_id`) and **Postgres LSN.**
- **Local client-side writes** to the SQLite database:
  - Are applied on top of the last checkpoint received from the server (so that querying SQLite includes local mutations not yet uploaded)
  - Are placed into the upload queue
- **The client retrieves the latest checkpoint from the server** *after it has finished uploading writes* (i.e. after the data is written to the Postgres database and the **client's upload queue is empty**)
  - The client then updates its local state to **match the server state.**

## System design, architecture & implementation

a. Design objectives
b. High-level architecture
c. Implementing *dynamic partial replication*
d. Syncing local writes back to Postgres
e. Server-authoritative architecture: No CRDTs needed
f. Guaranteeing consistency
**g. Guaranteeing data integrity**

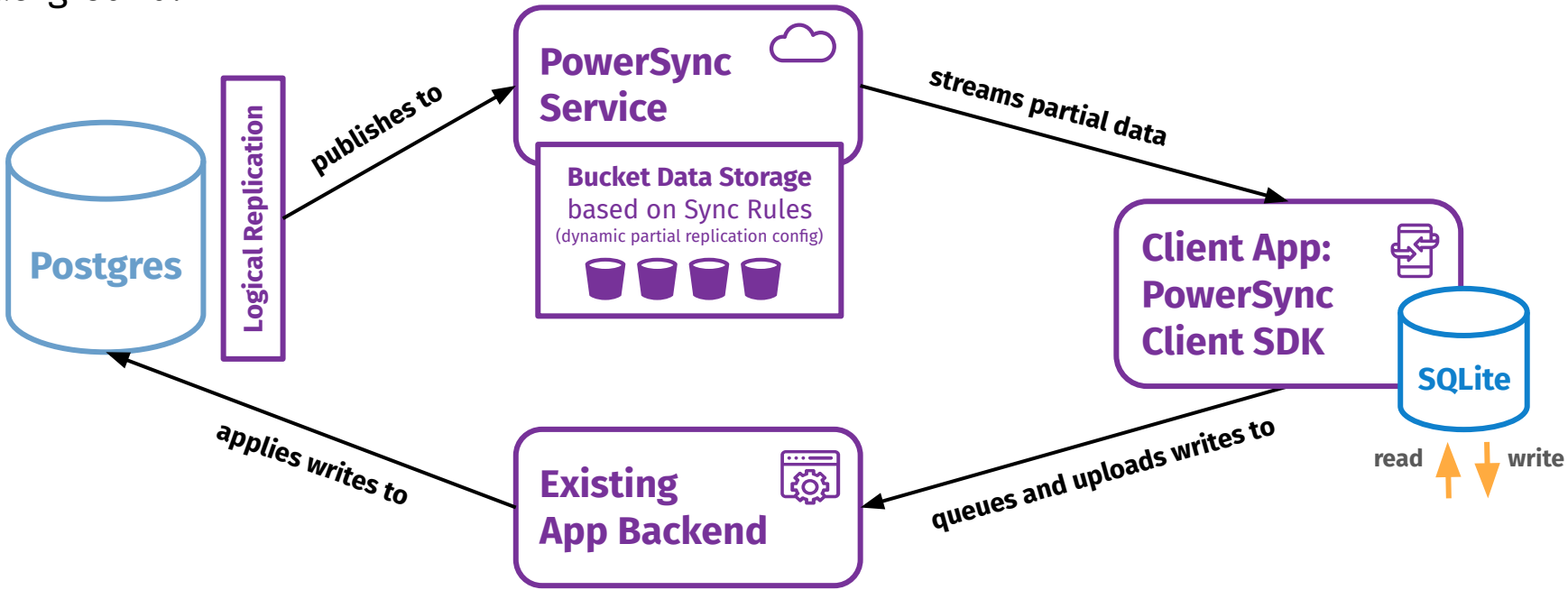
# Guaranteeing data integrity

- The client and server can compute a **per-bucket checksum.**
- Detecting data corruption: If the server and client checksums don't match, the client will re-download the bucket.

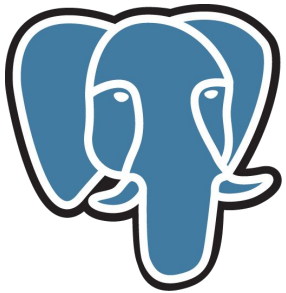


# Tying it all together

The sync system enables a **local-first app architecture**: The app reads/writes directly from/to a client-side SQLite database, which automatically syncs with a Postgres database in the background.



# Logical replication challenges & solutions

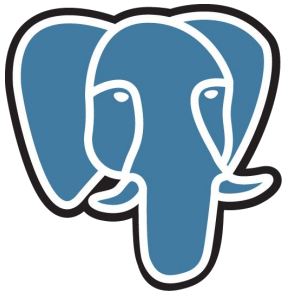


# Logical replication challenges & solutions

**Logical replication** was originally meant to be used for Postgres-to-Postgres replication in a way that is compatible over different Postgres versions and databases, replicating row data. (vs. physical replication: byte-by-byte)

**There were some challenges** to making the PowerSync model/architecture compatible with logical replication assumptions, for example:

1. Handling schema/DDL changes
2. LSNs that overlap between transactions
3. Accessing TOASTed values
4. Handling different row identifier permutations



# 1. Handling schema/DDL changes

What is provided with Postgres logical replication:

- **DDL changes do not get published** with Postgres logical replication.
- **Type identifiers** in messages can be used to identify column types.

Our design objective: We want the PowerSync system to be as **low-maintenance** as possible, including regarding schema changes.

# 1. Handling schema/DDL changes

We did a number of things in pursuit of this design objective:

- PowerSync protocol & client-side database storage are **schemaless**.
  - On the client, SQLite views are applied on top of schemaless JSON data.
- Sync Rules can apply some **transformations to Postgres data** during replication.
  - Developer can use this for some **backwards-compatibility:** Allowing clients to keep using an older schema.

*This reduces the surface area affected by schema changes.*

# 1. Handling schema/DDL changes

Various Postgres schema change scenarios are **detected wherever possible,** and either **automatically handled or documented** the manual action required:

| | | |
|---|---|---|
| `DROP` table | Developer action required | |
| `CREATE` table | Automatic | |
| `DROP` and re-`CREATE` table | Automatic | |
| `RENAME` table | Automatic | Re-sync the entire table |
| `REPLICA IDENTITY` changes | Automatic | Re-sync the entire table |
| Column changes | Developer action required | |
| Publication changes | Developer action required | |

There are some fundamental limitations, e.g. `GENERATED STORED` values are not published in the logical replication stream.

# 2. LSNs that overlap between transactions

**We need a monotonically increasing "operation ID"** (`op_id`) sequence, and initially used the Postgres LSNs to derive `op_id`. However, we moved away from that architecture to using an auto-incremented integer, for these reasons:

1. LSNs are not guaranteed to be monotonically increasing: Transaction commits are always in sequence in the WAL, but operations inside each transaction can have **LSN ranges overlapping** with other transactions.
    a. (For our checkpoint system, we keep track of the **commit LSNs**.)
2. We want to architecturally **support sharded databases** where each Postgres instance has its own LSN sequence that has to be merged into a singular `op_id` sequence
3. When we **reprocess sync data**, we need to continue with the same `op_id` sequence, while the LSN sequence does not get reset.

# 3. Accessing TOASTed values 🍞

- **TOAST (The Oversized Attribute Storage Technique):**
  If a value exceeds the page size (8 kB by default), it is compressed, and if still >= page size, it is stored in the TOAST table and a pointer is put in the original table.

- In the logical replication stream, a **TOASTed value is only published if the value has changed**. Otherwise, the field has an ‘unchanged’ flag.

- In order to be able to access current field values regardless of TOAST, we store an **additional copy of the current value** of any field that we replicate (*stateful stream processing*)

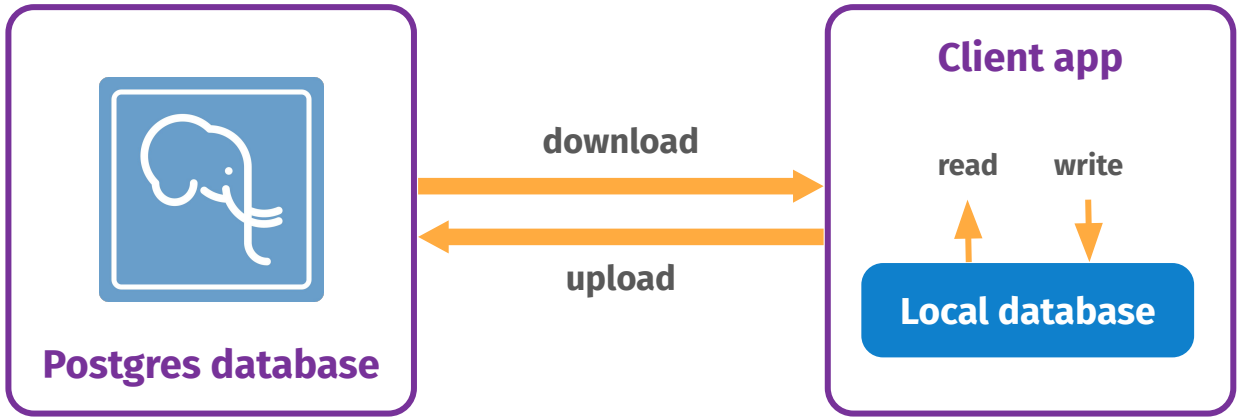
# 4. Handling different row identifier permutations

- Postgres has options in terms of **row identifier information** recorded in the logical replication stream, through `REPLICA IDENTITY`.
- We implemented support for all the different permutations for developer convenience (but we recommend `DEFAULT`, i.e. using the primary key)

| Replica Identity | Postgres Treatment | PowerSync Treatment |
|---|---|---|
| `DEFAULT` | Records primary key. | Use the primary key. |
| `FULL` | Records all columns. | Generate ID from all columns. |
| `USING INDEX` | Records columns referenced by named index. | Generate ID from columns referenced by named index. |
| `NOTHING` | No information recorded | Generate a random ID.<br>(Postgres blocks updating or deleting rows.) |

# We want to help advance local-first adoption

Local-first has several benefits for **developers and end-users.**




Our architecture provides for **Postgres<>anything** bidirectional sync

POWERSYNC

# Thank You

**Conrad Hofmeyr**
*Co-Founder & CEO, JourneyApps*

conrad@journeyapps.com
twitter.com/cahofmeyr
www.powersync.com