It's Not You, It's ~~Me~~ Your Tuples:

BREAKING UP WITH MASSIVE TABLES via PARTITIONING

Chelsea Dole
cdole@brex.com

- Staff Software Engineer, Brex
  - “The credit card for startups”, expense management software
  - Previously: Data Engineer, Backend Engineer
- Tech Lead, Data Storage Team
  - Postgres infrastructure
  - Query optimization
  - & more!

Chelsea Dole

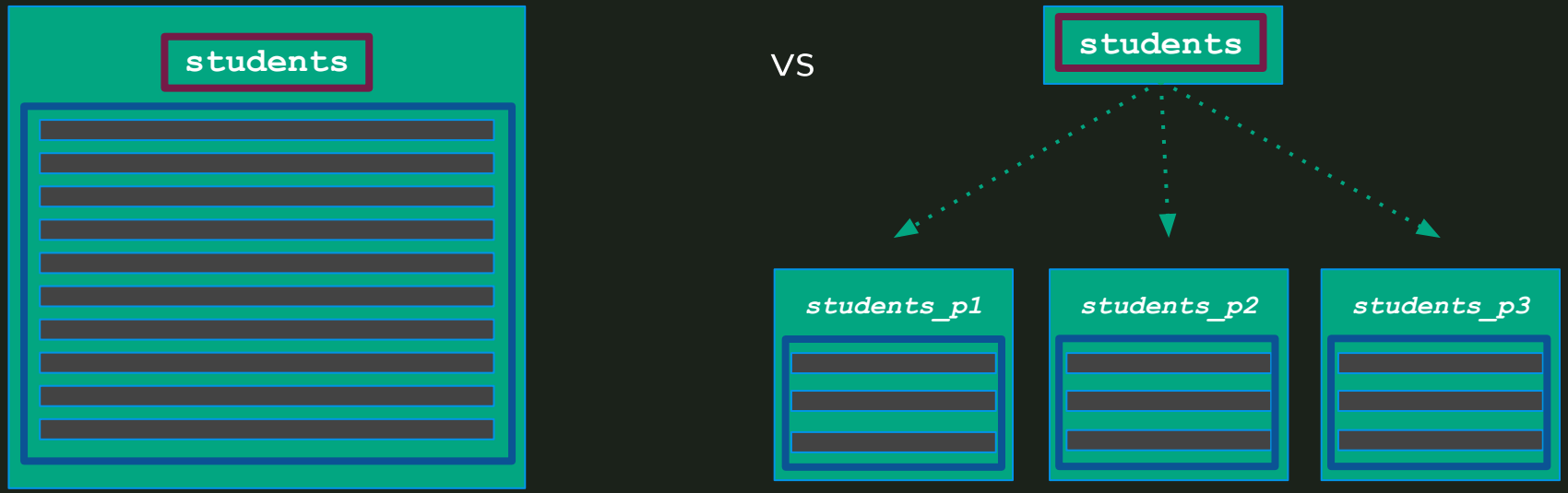
Outline

1. **What is partitioning?**
   a. Fundamentals of partitioning vs sharding
   b. The Postgres partitioning ecosystem
2. **Partitioning in Postgres**
   a. Types of partitioning: pros, cons, etc
3. **Why partition (or not)?**
   a. Actual vs perceived benefits of partitioning
   b. Gotchas
4. **How to partition an existing table**
   a. Methodology and trade-offs of four separate migration methods
5. **Maintenance, Configuration, & Observability**

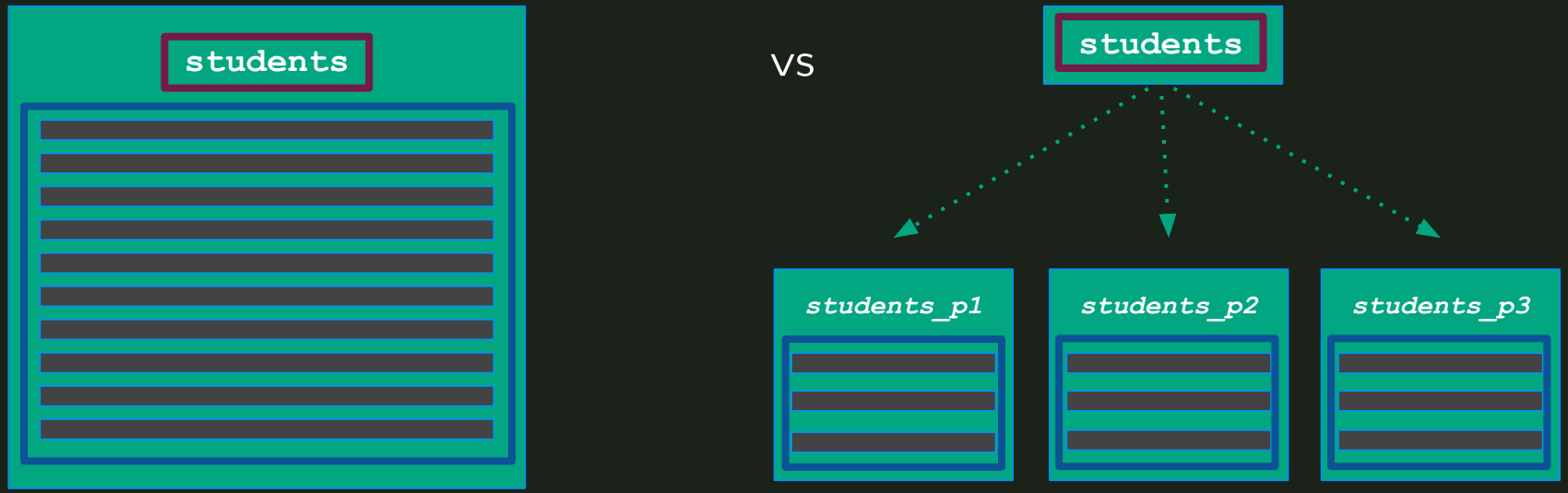
1. What is partitioning?

What is partitioning?

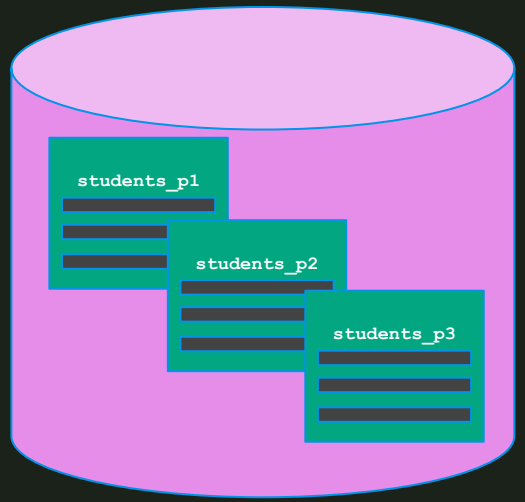
Splitting 1 larger, **logical table** into n smaller, **physical tables** [1]



```
SELECT id, full_name FROM students WHERE id = 1;
```



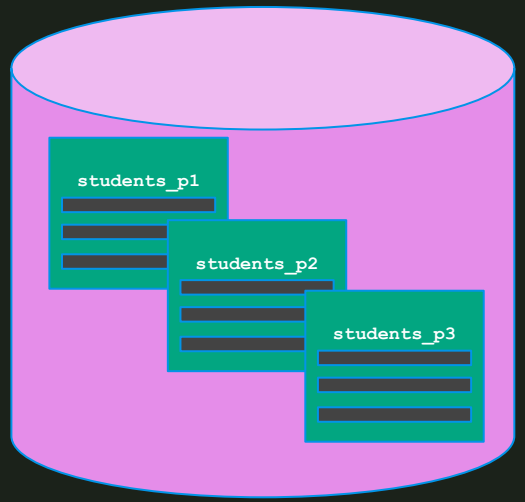
Sharding vs partitioning



Sharding: splitting 1 dataset across multiple nodes

(sometimes called “horizontal partitioning”)

Partitioning: splitting 1 dataset across multiple tables on the same node



Partitioning in Postgres

- **PG 9.6: partitioning via “table inheritance”**
  - Manual creation of “child tables”
  - Manual creation of triggers for `INSERT`s

Difficult setup, bad performance

- **PG 10: declarative partitioning**
  - Native support of partitioned tables:
    - `CREATE TABLE … PARTITION BY …`
  - `INSERT` “tuple routing” & pruning for `SELECT`s

Easy syntax, basic features

- **PG 11:**
  - Critical usability features for declarative partitioning
  - Default partition, hash type, `UPDATE` “tuple routing”, partition wise `JOIN`, & more

Solid features, broadly usable

Partitioning in Postgres

- **PG 12 - PG16+:**
  - Declarative partitioning performance & usability improvements, ex:
    - ATTACH/DETACH partition concurrently
    - Partition pruning improvements
    - Logical replication for partitioned tables
    - Reduced table locking on INSERT
    - & much more

Mature, first-class Postgres feature

2. Partitioning methods

1. Range
2. List
3. Hash

Partition key:

Defining how data is subdivided into partition tables

1. Range partitioning

- Partitions contain values within a predefined min/max
- Most common & useful method of partitioning

Examples:

- Time range data, mostly querying recent data
- Dashboard of “events”, preloading in chronological order

```
postgres=# CREATE TABLE students (
  id          BIGINT  NOT NULL,
  full_name   VARCHAR NOT NULL,
  school_name VARCHAR NOT NULL,
  grad_year   INTEGER NOT NULL,
  inserted_at TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id, inserted_at)
) PARTITION BY RANGE(inserted_at);

postgres=# CREATE TABLE students_09_2023 PARTITION OF students
FOR VALUES FROM ('2023-09-01 00:00:000') TO ('2023-09-30 23:59:999');

postgres=# CREATE TABLE students_10_2023 PARTITION OF students
FOR VALUES FROM ('2023-10-01 00:00:000') TO ('2023-10-31 23:59:999');

<...>
```

2. List partitioning

- Partitioning based on explicit column value options
- Low cardinality values & `DEFAULT` partition
- Skewed partition table size

Examples:

- Data separated by user region (EX: “eu”, “apac”, etc)
- Data may be bulk loaded/dropped by list partition
- Potential values for PK do not change dynamically

```
postgres=# CREATE TABLE students (
  id          BIGINT  NOT NULL,
  full_name   VARCHAR NOT NULL,
  school_name VARCHAR NOT NULL,
  grad_year   INTEGER NOT NULL,
  inserted_at TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id, grad_year)
) PARTITION BY LIST(grad_year);

postgres=# CREATE TABLE students_2023 PARTITION OF students
FOR VALUES IN (2023);

postgres=# CREATE TABLE students_2024 PARTITION OF students
FOR VALUES IN (2024);

postgres=# CREATE TABLE students_default PARTITION OF students DEFAULT;
```

3. Hash partitioning

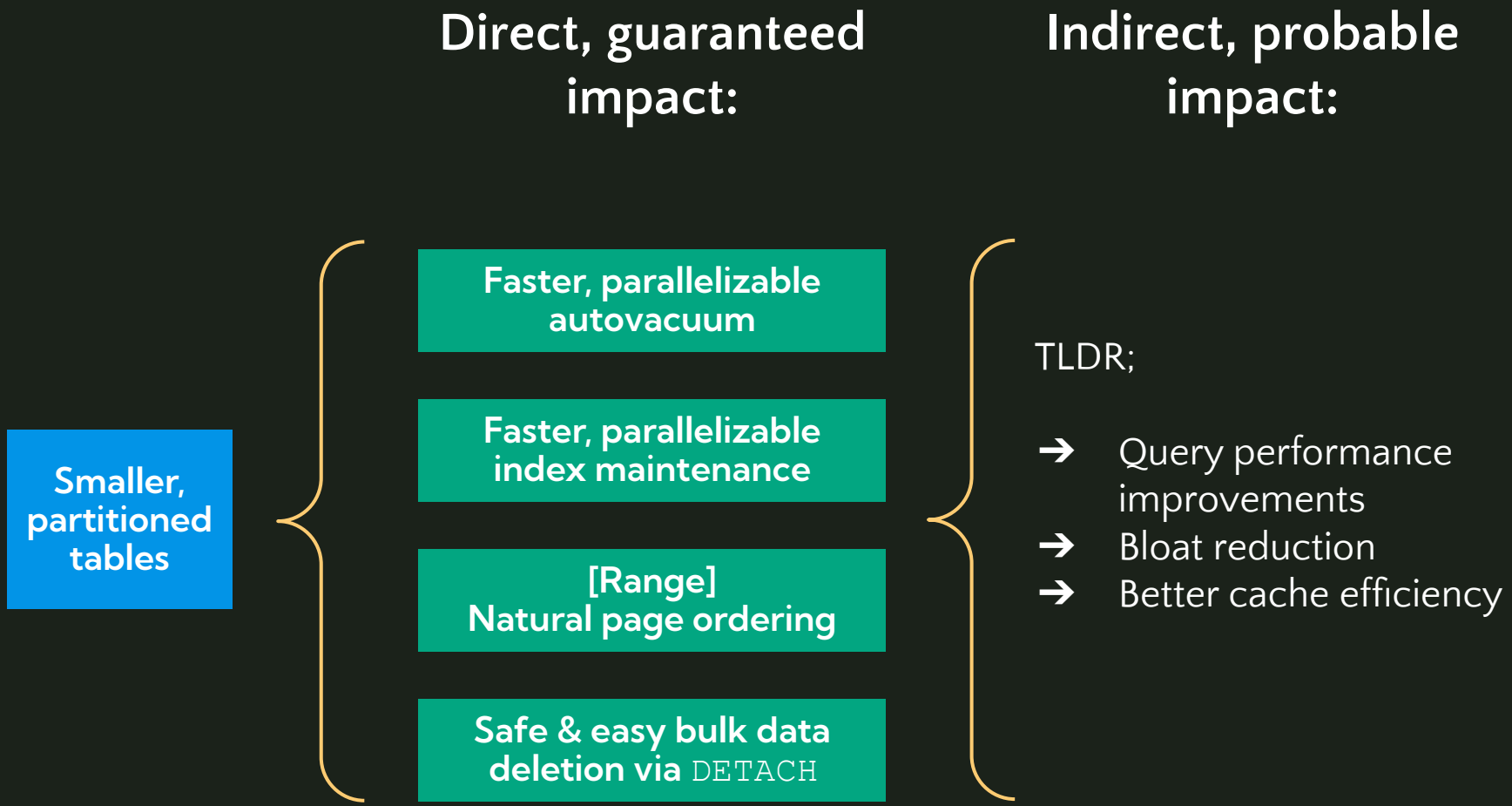
- Partitioning based on a hashed column value, defining `MODULUS` & `REMAINDER`
- Usually used to distribute values evenly across smaller tables when there is no “natural” partition key

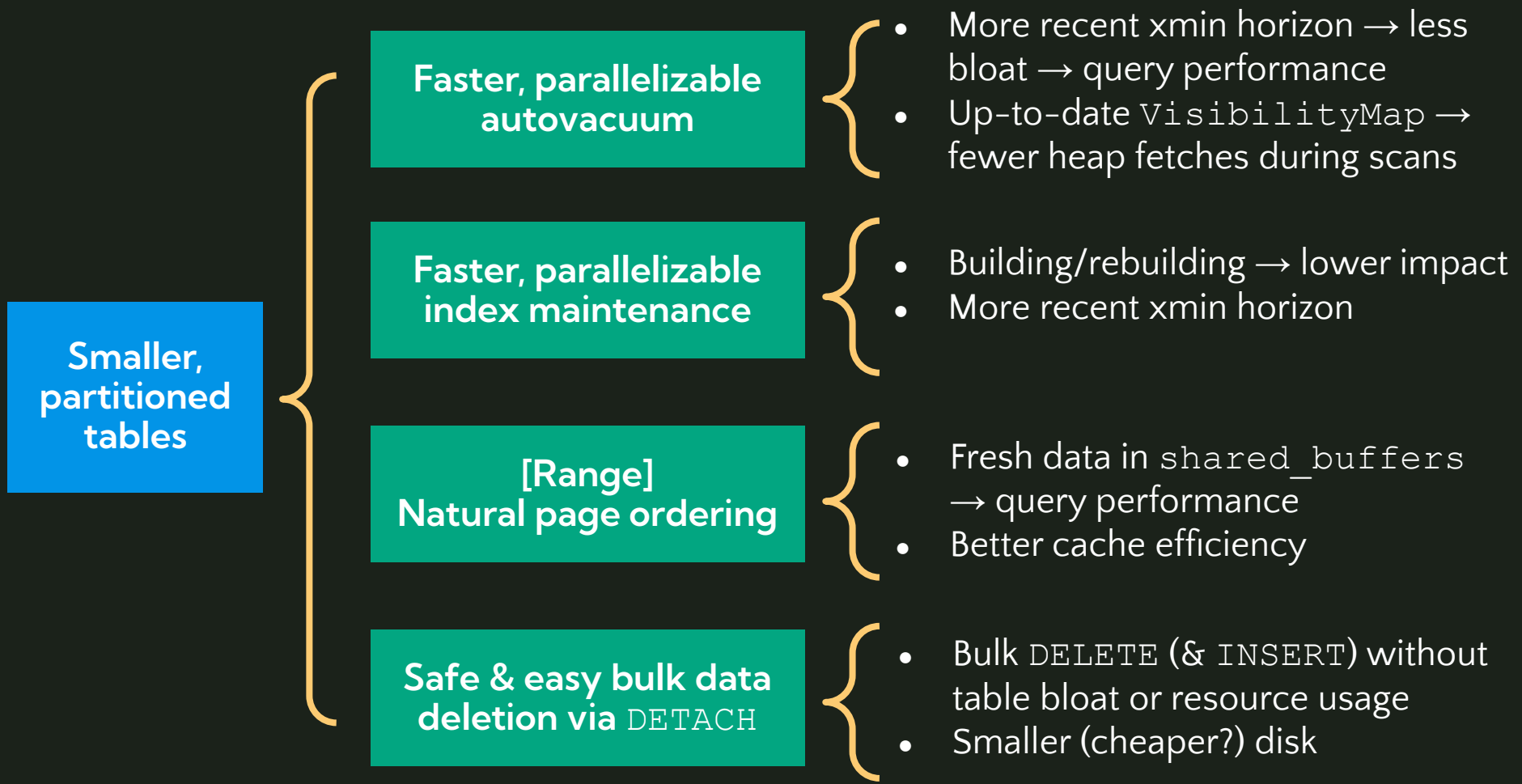
Examples:

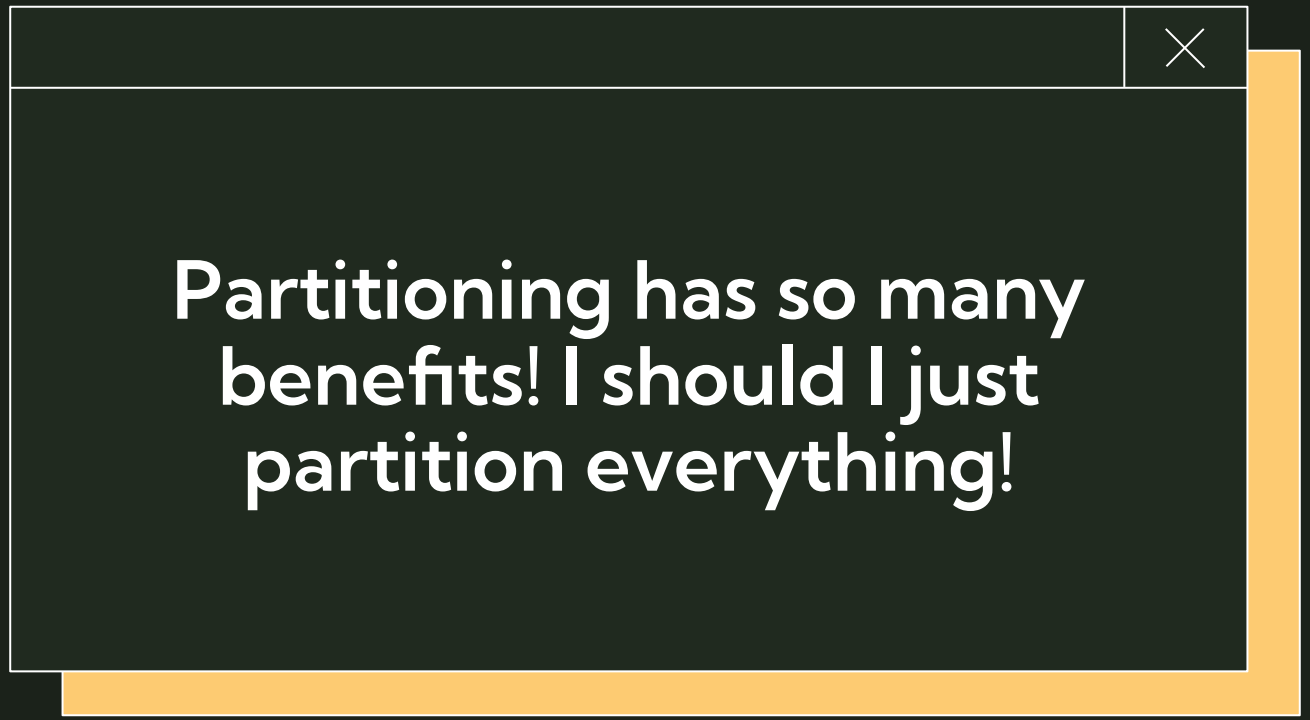
- Partitioning is necessary for table maintenance/health, but there is no natural partition key

```
postgres=# CREATE TABLE students (
  id          BIGINT  PRIMARY KEY,
  full_name   VARCHAR NOT NULL,
  school_name VARCHAR NOT NULL,
  grad_year   INTEGER NOT NULL,
  inserted_at TIMESTAMPTZ NOT NULL
) PARTITION BY HASH(id);

postgres=# CREATE TABLE students_0 PARTITION OF students FOR VALUES
WITH (MODULUS 3, REMAINDER 0);

postgres=# CREATE TABLE students_1 PARTITION OF students FOR VALUES
WITH (MODULUS 3, REMAINDER 1);

postgres=# CREATE TABLE students_2 PARTITION OF students FOR VALUES
WITH (MODULUS 3, REMAINDER 2);
```

3. Why partition (or not)?


Direct, guaranteed impact:
Indirect, probable impact:
Smaller, partitioned tables
Faster, parallelizable autovacuum
Faster, parallelizable index maintenance
[Range] Natural page ordering
Safe & easy bulk data deletion via DETACH
TLDR;
➔ Query performance improvements
➔ Bloat reduction
➔ Better cache efficiency


Smaller, partitioned tables
Faster, parallelizable autovacuum
More recent xmin horizon → less bloat → query performance
Up-to-date VisibilityMap → fewer heap fetches during scans
Faster, parallelizable index maintenance
Building/rebuilding → lower impact
More recent xmin horizon
[Range] Natural page ordering
Fresh data in shared_buffers → query performance
Better cache efficiency
Safe & easy bulk data deletion via DETACH
Bulk DELETE (& INSERT) without table bloat or resource usage
Smaller (cheaper?) disk


Partitioning has so many benefits! I should I just partition everything!

Partitioning has so many benefits! I should I just partition everything!

Downsides of partitioning

- Possible negative impact on performance
  - Bad performance on queries without partition key
  - Increased query planning time with high partition count
    - This downside is drastically reduced in recent version of PG
- Stronger Postgres knowledge required from app developers & product
  - Understanding the impact of writing queries without partition key
  - Postgres becomes less of a “generic, all-purpose tool”
- Advanced features → advanced expertise
  - Postgres “partitioning ecosystem” requires more bespoke knowledge
  - Advanced observability, knowledge of “gotchas”, extensions, etc

When is partitioning “worth it”?

Industry rule-of-thumb:

- Table size >=100GB ⭐

Postgres docs:

- Table size > physical memory of the server

When is partitioning "worth it"?

My (far less official) rules-of-thumb:

`RANGE` partitioning:

- Easily the best method/return on value
- If your table has a "natural" range partition key or if you want to "expire" old data, do it

`LIST` partitioning:

- If you need to regularly bulk `DELETE` or `INSERT` data for a new list option

`HASH` partitioning:

- Partitioning is needed for maintenance reasons, but there's no natural PK
- There are no plans to `ATTACH`/`DETACH` partitions

Downsides of partitioning

- Possible negative impact on performance
  - Bad performance on queries without partition key
  - Increased query planning time with high partition count
    - This downside is drastically reduced in recent version of PG
- Stronger Postgres knowledge required from app developers & product
  - Understanding the impact of writing queries without partition key
- Advanced features → advanced expertise
  - Postgres "partitioning ecosystem" requires more bespoke knowledge
  - Advanced observability, **knowledge of "gotchas",** extensions, etc

The Big Gotcha

Table primary keys & unique constraints must include the partition key

```
ERROR: insufficient columns in PRIMARY KEY constraint
definition

    PRIMARY KEY constraint on table "students" lacks column
    "inserted_at" which is part of the partition key.
```

range partitioning setup

```
postgres=# CREATE TABLE students (
  id           BIGINT  NOT NULL,
  full_name    VARCHAR NOT NULL,
  school_name  VARCHAR NOT NULL,
  grad_year    INTEGER NOT NULL,
  inserted_at  TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id, inserted_at)
) PARTITION BY RANGE(inserted_at);

postgres=# CREATE TABLE students_09_202
FOR VALUES FROM ('2023-09-01 00:00:000'

postgres=# CREATE TABLE students_10_202
FOR VALUES FROM ('2023-10-01 00:00:000'

<...>
```

What if the source table already defines PK, but it's not my desired partition key?

Migrate PRIMARY KEY to a composite key , ex: `(id, inserted_at)`

- Beware of `UPSERT`s, which need to provide all primary key fields
- In this case, `id` is no longer individually UNIQUE

Rapid Fire Gotchas

- `RANGE` & `LIST` partitioning:
  - `DEFAULT` partition – feature or bug?
- `HASH` partitioning:
  - Range queries (i.e., `WHERE <partition_key> BETWEEN x, y`) can't use partition pruning
  - Partition count cannot be changed without re-partitioning
- Logical replication/CDC
  - Before PG13, logical replication was not supported for partitioned tables
  - `publish_via_partition_root`

4. Partitioning an existing table

Why is this a challenge?

- Typically, existing tables are migrated to be partitioned, rather than starting as partitioned
- Declarative partitioning doesn't support "`ALTER TABLE ... PARTITION BY`" syntax, so this migration must be performed manually

Four examples, four (of many!) ways to partition:

1) Use Case #1: offline migration
2) Use Case #2: online migration (duplicating disk space)
3) Use Case #3: online migration (no duplicated disk space)
4) Use Case #4: online migration (logical replication)

‼️ Disclaimer

There are MANY ways to partition tables. This talk is relatively technology agnostic – so examples focus on “native Postgres” methods which I’ve used, rather than diving deep into specific extensions

- pg_partman
- pgslice
- pg_party

Extensions which provide various partitioning migration utilities, among other functionality

Use Case #1: Offline migration

At the start of each school year, admins insert ~500K students for the new `grad_year`, and delete ~500K newly-graduated students.

- 100GB table `students` serves live traffic
  - 90% read, 10% insert/update/delete
- Traffic is concentrated during 9am-5pm M-F
- School pays teachers really well, no DBA budget

Desired Schema

```
CREATE TABLE students(

  <...>

) PARTITION BY
LIST(grad_year);
```

Constraints:

- ✅ <=3 hours scheduled downtime acceptable
- ✅ 200GB disk space available

```
-- Step #1: Create a new, list partitioned table with the same schema &
indexes as "students", and create partitions for the empty table

postgres=# CREATE TABLE s_v2(
  id         BIGINT NOT NULL,
  <...>
  grad_year INTEGER NOT NULL,
  PRIMARY KEY(id, grad_year)
) PARTITION BY LIST(grad_year);

postgres=# CREATE TABLE students_2014 PARTITION OF s_v2 FOR VALUES IN
(2014);

<...>

postgres=# CREATE TABLE students_2023 PARTITION OF s_v2 FOR VALUES IN
(2023);

postgres=# CREATE INDEX students__grad_year ON s_v2 (grad_year);
```

```
-- Step #2: Manually insert the data, though your preferred means:
  -  - INSERT - (example below), very large insert, unbatched
  -  - pg_partman[1] - this is a great place to use pg_partman's
                       partman.partition_data_proc() function, as it
                       will batch INSERTs natively
  -  - pg_dump/load

postgres=# BEGIN;

postgres=# INSERT INTO s_v2 (
    SELECT * FROM students
);

    <...>
```

[1] https://github.com/pgpartman/pg_partman/blob/master/doc/pg_partman_howto.md#offline-partitioning

```
-- Step #3: Within in the same transaction, “swap” the two tables

    <...>

postgres=# ALTER TABLE students RENAME TO students_archived;

postgres=# ALTER TABLE s_v2 RENAME TO students;

postgres=# COMMIT;

-- Step #5: Now back online, drop the unpartitioned
“students_archived” table, freeing up disk space

postgres=# DROP TABLE students_archived;
```

Use Case #2: Online migration, duplicating tables

The school district is running into issues with DB maintenance time (vacuum, reindexing, etc), and expects 2x data growth this year due to districts merging. Read queries filters vary significantly.

- 300GB table `students` serves live traffic
  - 60% read, 30% insert/update/delete
- Traffic is evenly distributed throughout the day

Constraints:

- ⚠️ <=3m downtime acceptable
- ✅ 500GB disk space available

Desired Schema

```
CREATE TABLE students(
  id uuid PRIMARY KEY,

  <...>

) PARTITION BY
HASH(id);
```

```
-- Step #1: Create a new, hash partitioned table with the same
schema as "students", and create partitions for the empty table

postgres=# CREATE TABLE s_v2 (
    LIKE students
    INCLUDING DEFAULTS INCLUDING INDEXES INCLUDING CONSTRAINTS
) PARTITION BY HASH(id);

postgres=# CREATE TABLE students_0 PARTITION OF s_v2 FOR VALUES WITH
(MODULUS 10, REMAINDER 0);

<...>

postgres=# CREATE TABLE students_9 PARTITION OF s_v2 FOR VALUES WITH
(MODULUS 10, REMAINDER 9);
```

```
-- Step #2: Create a plpgsql function returning a trigger which duplicates
incoming INSERT/UPDATE/DELETE operations to s_v2

postgres=# CREATE OR REPLACE FUNCTION duplicate_to_partitioned_table()
RETURNS TRIGGER AS
$$
BEGIN
    <...>
END;
$$ LANGUAGE PLPGSQL;

-- Step #3: Create a trigger, so the function is called after
INSERT/UPDATE/DELETE on the "students" table.

postgres=# CREATE TRIGGER duplicate_to_partitioned_table_trigger
AFTER INSERT OR UPDATE OR DELETE ON students
FOR EACH ROW EXECUTE PROCEDURE duplicate_to_partitioned_table();
```

https://bit.ly/data-duplication-partitioning-gist

```
-- Step #4: Run a historical backfill for data from “students”, inserting into s_v2 in batches. When primary key conflicts are found, do nothing.

-- Step #5: Swap the tables in place, then drop the old table

postgres=# BEGIN;

    ALTER TABLE students RENAME TO students_archived;

    ALTER TABLE s_v2 RENAME TO students;

COMMIT;

postgres=# DROP TABLE students_archived;
```

Use Case #3: Online migration, no table duplication

The school district is running into issues with DB maintenance time (vacuum, reindexing, etc), and expects 2x data growth this year due to districts merging. Read queries filters vary significantly.

- 300GB table `students` serves live traffic
  - 60% read, 30% insert/update/delete
- Traffic is evenly distributed throughout the day

Desired Schema

```
CREATE TABLE students(

  <...>

) PARTITION BY
HASH(id);
```

Constraints:

- ⚠️ <=3m downtime acceptable
- ⚠️ 100GB disk space available

Not enough disk space available on the server to duplicate the dataset

```
-- Step #1: Create a new, hash partitioned table with the same
schema as "students", and create partitions for the empty table

postgres=# CREATE TABLE s_v2 (
    LIKE students
    INCLUDING DEFAULTS INCLUDING INDEXES INCLUDING CONSTRAINTS
) PARTITION BY HASH(id);

postgres=# CREATE TABLE students_0 PARTITION OF s_v2 FOR VALUES WITH
(MODULUS 10, REMAINDER 0);

<...>

postgres=# CREATE TABLE students_9 PARTITION OF s_v2 FOR VALUES WITH
(MODULUS 10, REMAINDER 9);
```

```
-- Step #2: Create a plpgsql function returning a trigger which:
      - ON INSERT: inserts only to new table
      - ON DELETE: deletes from both new & old table
      - ON UPDATE: deletes from old table, inserts or updates new table

postgres=# CREATE OR REPLACE FUNCTION migrate_to_partitioned_table()
RETURNS TRIGGER AS
$$
BEGIN

    <...>

END;
$$ LANGUAGE PLPGSQL;
```

https://bit.ly/data-migration-partitioning-blog[1]

[1] *"Partitioning a large table without a long-running lock", 2ndQuadrant (Andrew Dunstan)*

```
-- Step #3: Replace the old "students" table with a view which UNIONs the old
and new table results together. Then create a trigger which calls
migrate_to_partitioned_table() *INSTEAD OF* (not after) INSERT/UPDATE/DELETE to
the "students" table.

postgres=# BEGIN;

    ALTER TABLE students RENAME TO students_archived;

    CREATE VIEW students AS
         SELECT id, data FROM students
         UNION ALL
         SELECT id, data FROM s_v2
    ;

    CREATE TRIGGER migrate_to_partitioned_table_trigger
    INSTEAD OF INSERT OR UPDATE OR DELETE on students
    FOR EACH ROW
    EXECUTE FUNCTION migrate_to_partitioned_table();

COMMIT;
```

```
-- Step #4: Run a backfill to insert rows from the old table to the new,
partitioned table. (Example available in previous 2ndQuadrant blog link.)

-- Step #5: Drop the view and migration function. Rename the new,
partitioned table to be “students”. In a separate transaction, drop the old
“students_archived” table

postgres=# BEGIN;
    DROP VIEW students;
    DROP FUNCTION migrate_to_partitioned_table();
    ALTER TABLE s_v2 RENAME TO students;
COMMIT;

postgres=# DROP TABLE students_archived;
```

Use Case #4: Logical replication 🚀📈

For reasons indecipherable, the `students` table is >1TB. The district expects to regularly partition more tables, so they want the process to be repeatable. Apps connect to the PGBouncer DNS name (i.e, `students-pgbouncer.io:5432`) rather than the "real" host name.

- 1.2TB table `students` serves live traffic
  - 80% read, 20% insert/update/delete
- Traffic is evenly distributed throughout the day

Desired Schema

```
CREATE TABLE students(

  <...>

) PARTITION BY
RANGE(inserted_at);
```

Constraints:

- ⚠️ <=3m write downtime acceptable
- ⚠️ 100GB disk space available
- ⚠️ Task must be easily repeatable

Use Case #4: Logical replication 🚀 📈

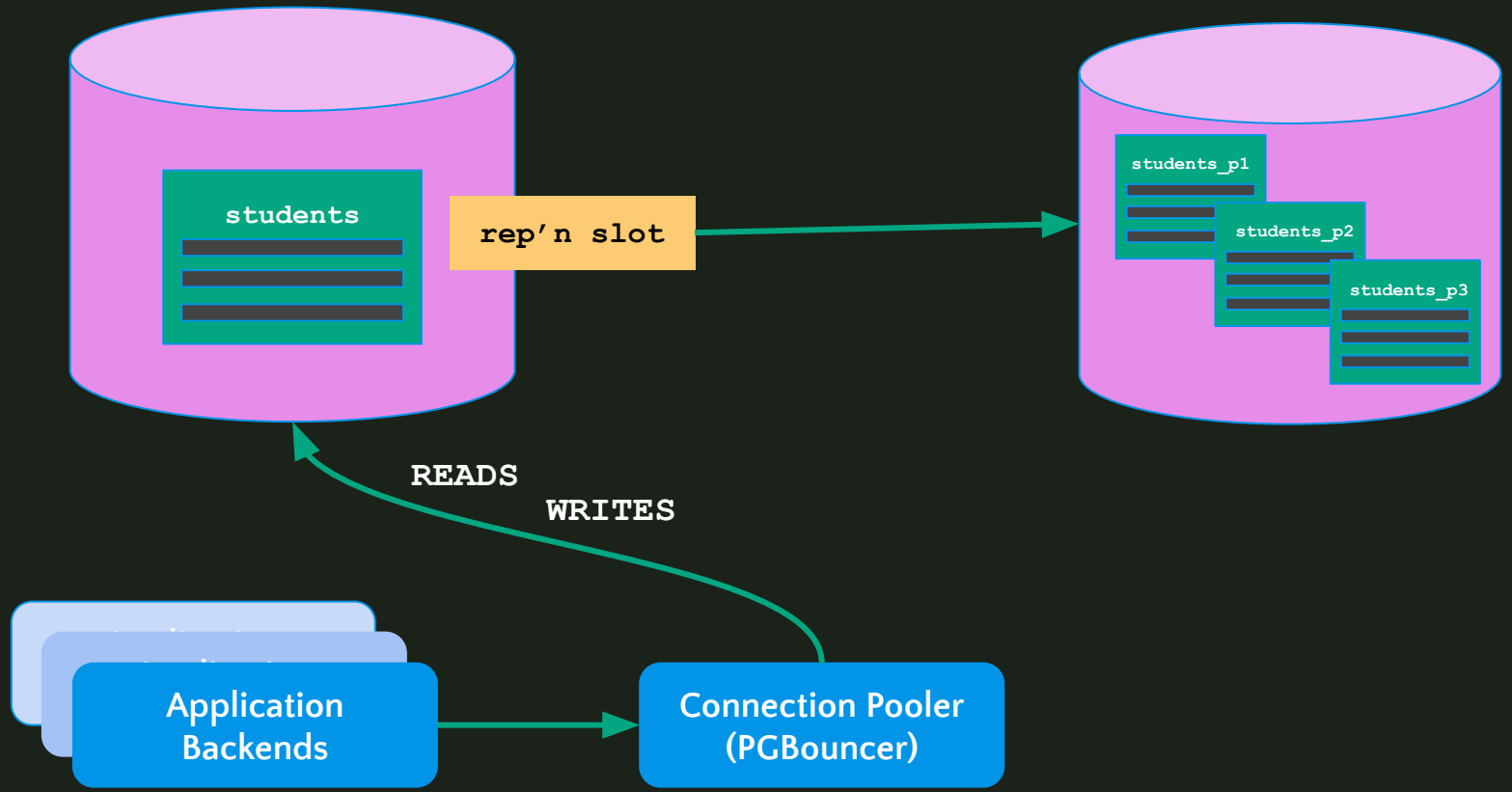
TLDR;

1) Ensure the source `students` table has a primary key, and don't use `SEQUENCE`s
2) Set up a new, fresh database instance, and create the desired partitioned schema & database roles there
3) Create a `PUBLICATION` for the table(s) on the source DB
4) Create a `SUBSCRIPTION` on the new destination DB
5) Wait for logical replication to catch up
    a) Ensure no DDL migrations occur
6) Migrate any replica/explicitly read-only connections from the source to the destination
7) Cut off writes to the primary by scaling down PGBouncer to 0
8) Check replication slot lag/`LSN` to ensure all data is transferred
9) Re-configure PGBouncer to point at the new, destination DB
10) Scale PGBouncer back up

Writes Offline

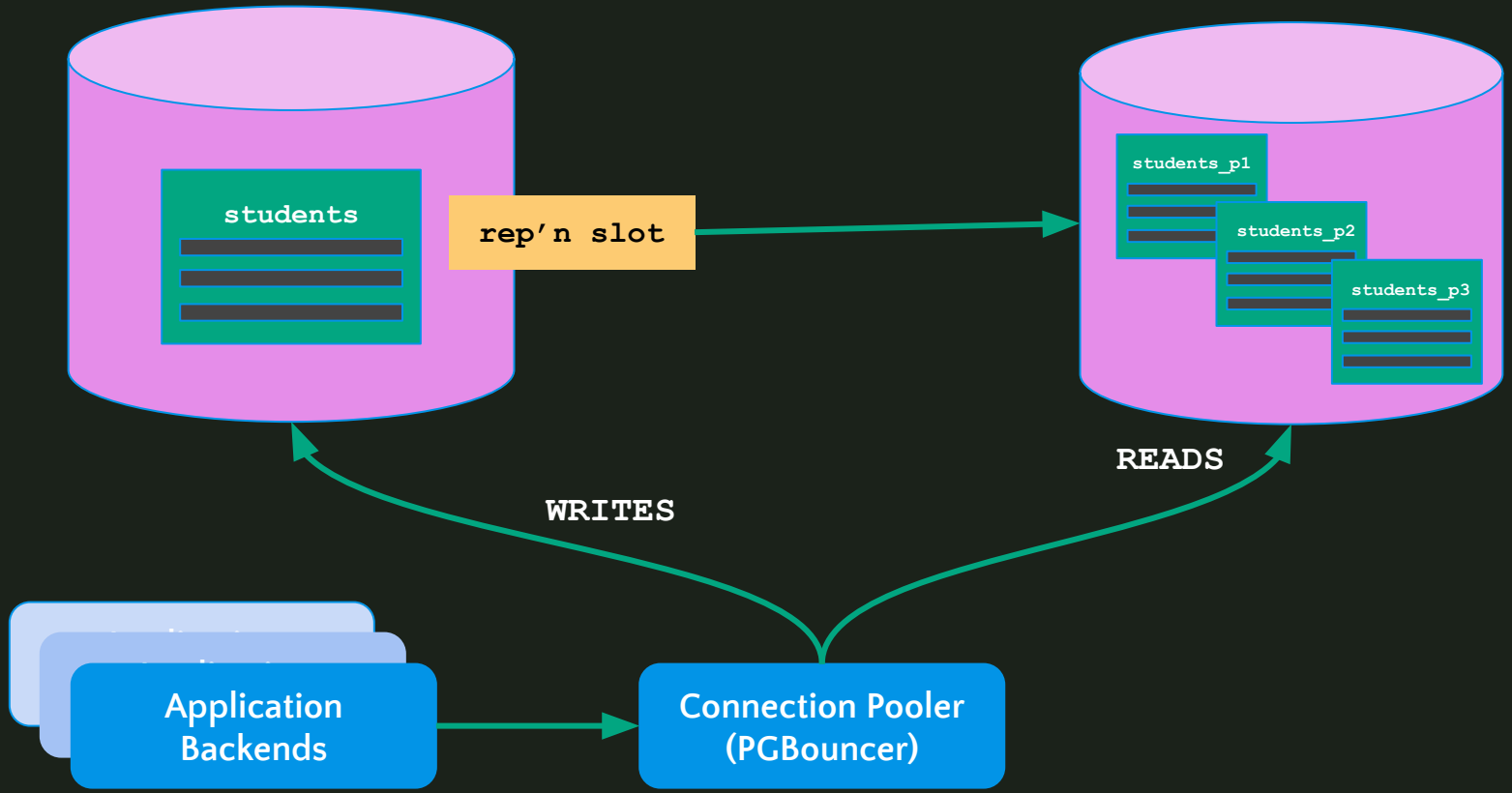
Step 1

Use Case #4: Logical replication 🚀 📈



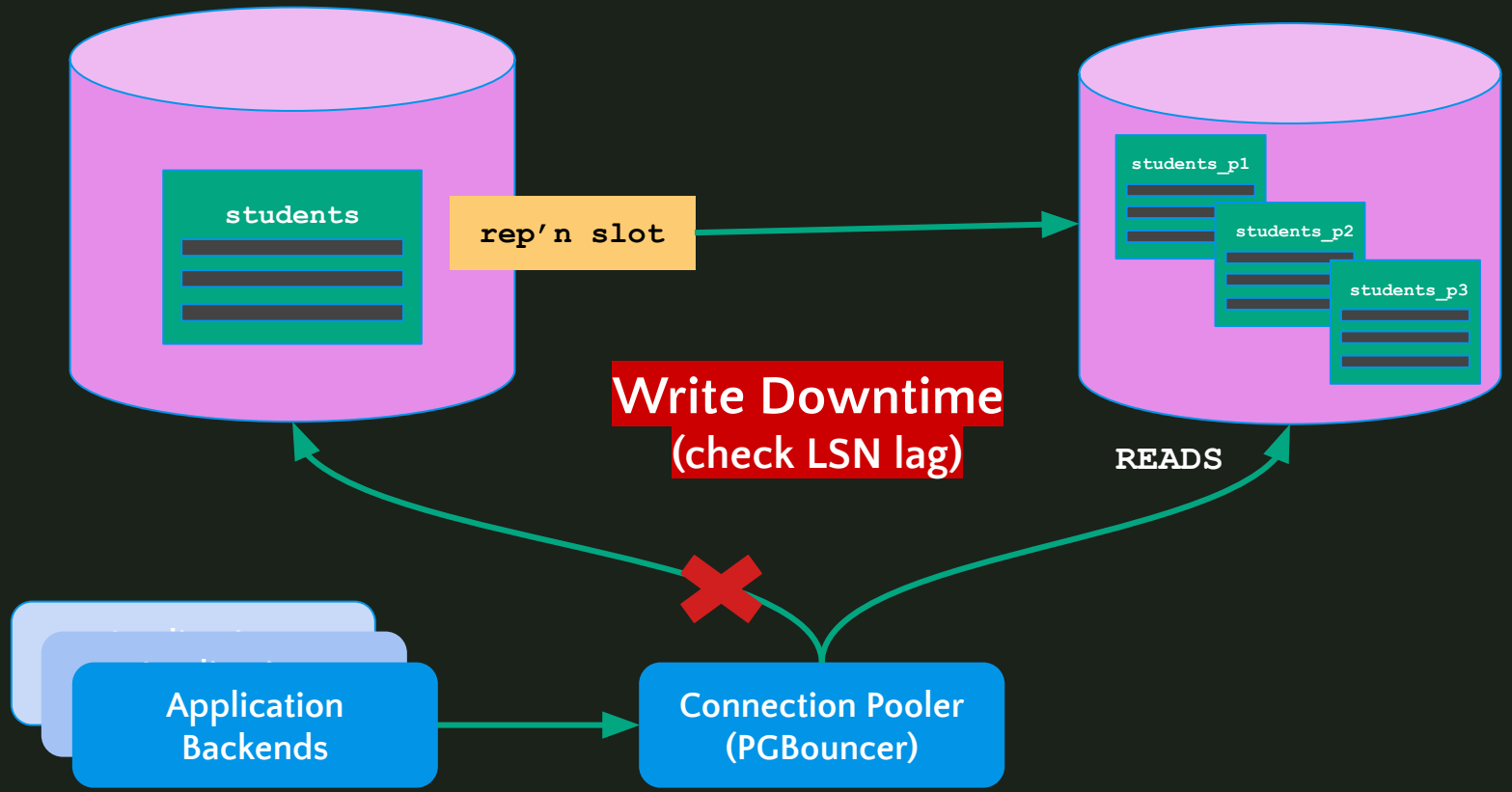
Step 2

Use Case #4: Logical replication 🚀 📈



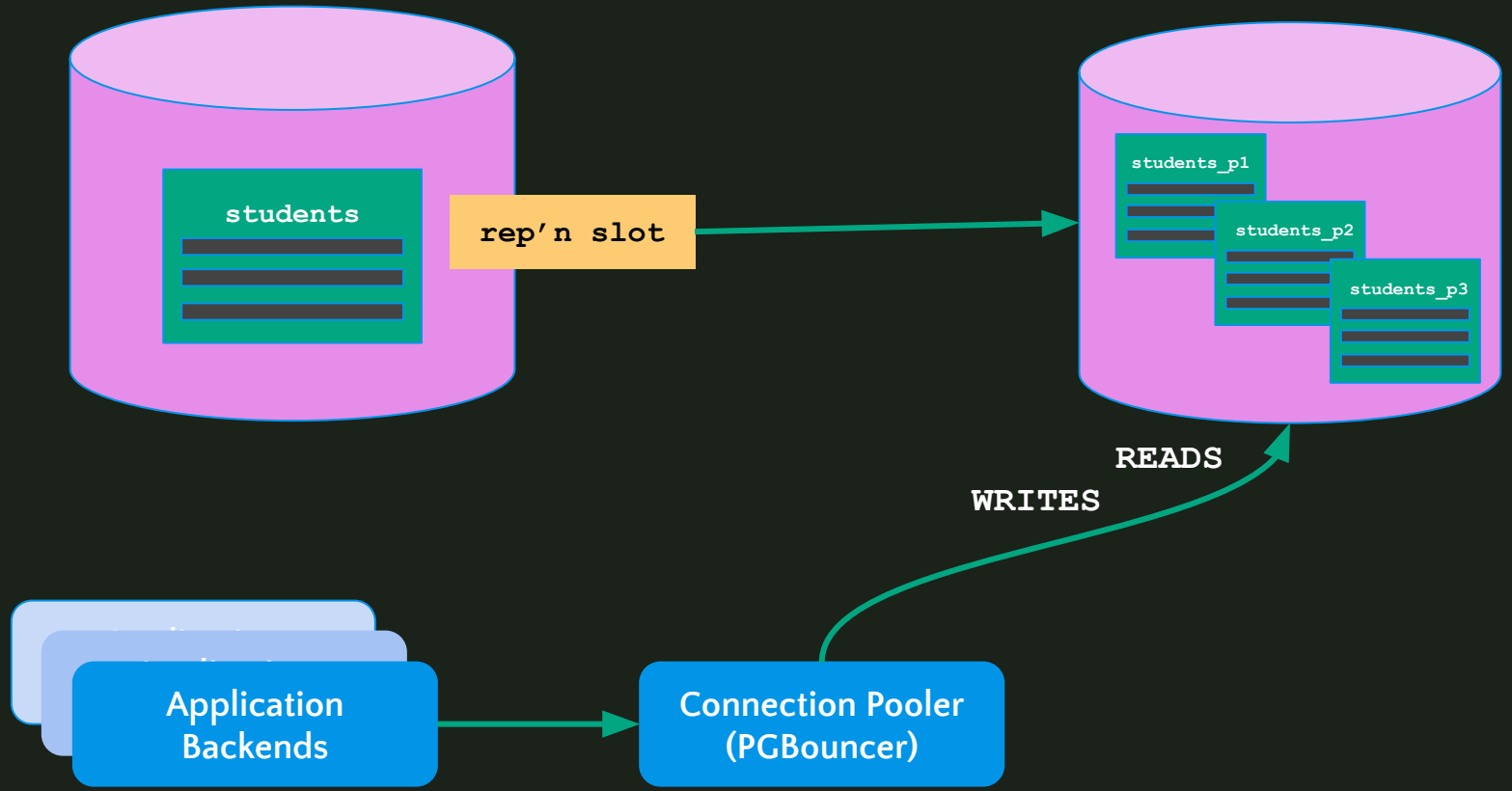
Step 3

Use Case #4: Logical replication 🚀📈



Step 4

Use Case #4: Logical replication 🚀 📈



Use Case #4: Logical replication 🚀 📈

TLDR;

1) Set up a **new, fresh database instance**, and create the desired partitioned schema & identical roles there
   a) Schema may contain JUST "students", or all tables
2) Create a `PUBLICATION` for the table(s) on the
3) Create a `SUBSCRIPTION` on the new destinat
4) Wait for logical replication to catch up
5) Migrate any replica/explicitly read-only connections t
   source to the destination
6) Cut off writes to the primary by scaling down P
7) Check replication slot lag/`LSN` to ensure all da
8) Re-configure PGBouncer to point at the new, d
9) Scale PGBouncer back up
10) Profit

But wait... do I even need to publish to a new instance? Can I just logically replicate from "students" to a second, partitioned table in the same DB/instance?

Use Case #4: Logical replication 🚀 📈

TLDR;

1) Set up a new, fresh database instance, and cr
   partitioned schema & identical roles there
   a) Schema may contain JUST “students”, or all table
2) Create a `PUBLICATION` for the table(s) on th
3) Create a `SUBSCRIPTION` on the new destina
4) Wait for logical replication to catch up
5) Migrate any replica/explicitly read-only connections t
   source to the destination
6) Cut off writes to the primary by scaling down P
7) Check replication slot lag/`LSN` to ensure all da
8) Re-configure PGBouncer to point at the new, d
9) Scale PGBouncer back up
10) Profit

Schema, table, and columns names must be the same on the `PUBLISHER` and `SUBSCRIBER`.

- If instance stays the same, DB name must change
- Changing instances == “free” upgrade 🤷‍♀️

4. Maintenance, Configuration, Observability, etc

Maintenance

- Regular creation of new partitions
  - `RANGE: pg_partman`
  - `LIST: pg_partman`, migrations

`pg_partman`:

An extension to create and manage both time-based and number-based table partition sets.

- Automatically create or detach/delete old partitions
- `CALL partman.run_maintenance_proc(<...>);`

Observability

Monitoring/alerting:

- Partitions are created/deleted by `pg_partman` as expected
  - Alert with lack of data, not just explicit failures
- Partition size (skew) – especially for list partitions

`auto_explain`:

- Dynamically help detect slow query plans, likely not including partition key

Configuration

Any configuration changes are made on the basis of table count growing. The fact that the tables are partitions isn't important.

`autovacuum_max_workers` (default=3)

- Consider increasing, based on on resource usage

Organizational Support

Building an understanding of partitioning & its benefits/constraints across the engineering organization. EX:

- Internal/open source blog posts
- Git hooks linking documentation for schema migrations or new queries

TLDR;

- How can your partitioned table(s) stay performant and well-understood going forward?
- How can you enable engineers to write partitioning-aware queries?

Thank you!

Chelsea Dole

cdole@brex.com
https://www.linkedin.com/in/chelsea-dole/