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

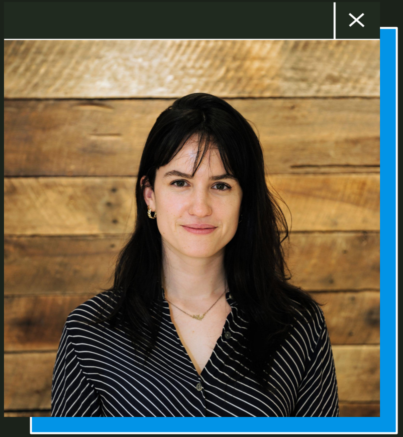
BREAKING UP MASSIVE TABLES via PARTITIONING

Chelsea Dole

- Database Engineer, *financial services*
- Organizer, *PGSummit US (PGConf NYC)*

Previously…

➔ Staff Database Engineer, *Brex*
➔ Data Engineer, *Coffee Meets Bagel*
➔ Etc



Chelsea Dole

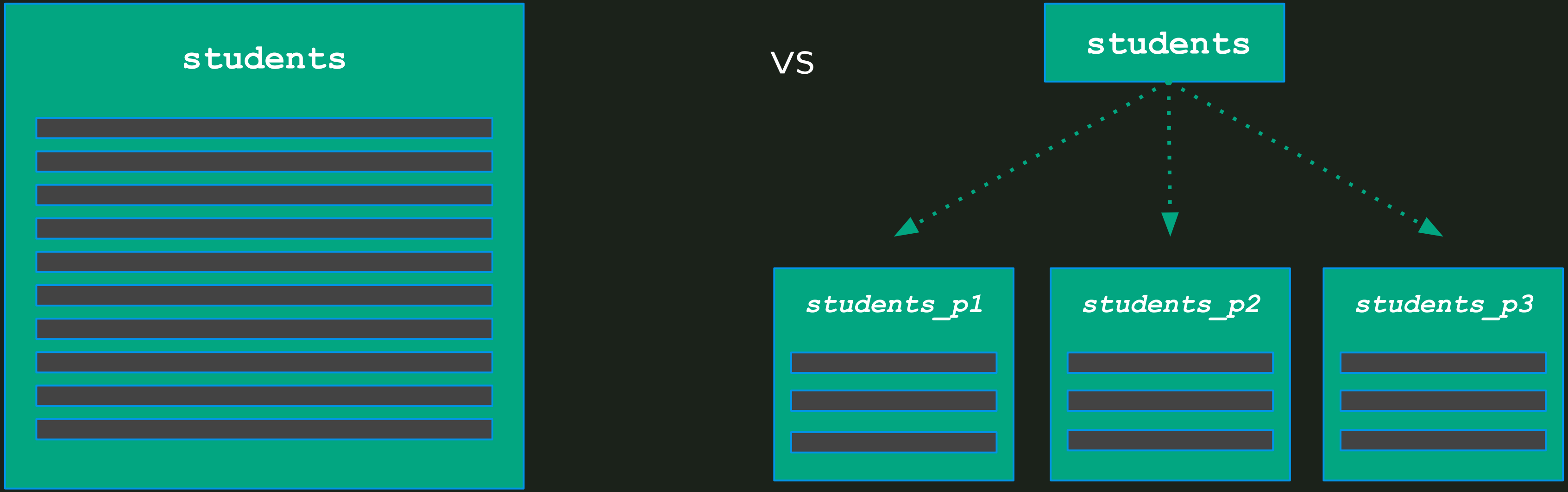
Outline

1. What is partitioning?
2. Partitioning in Postgres
3. Why partition (or not)?
4. How to partition an existing table
5. Maintenance, configuration, & observability

1. What is partitioning?

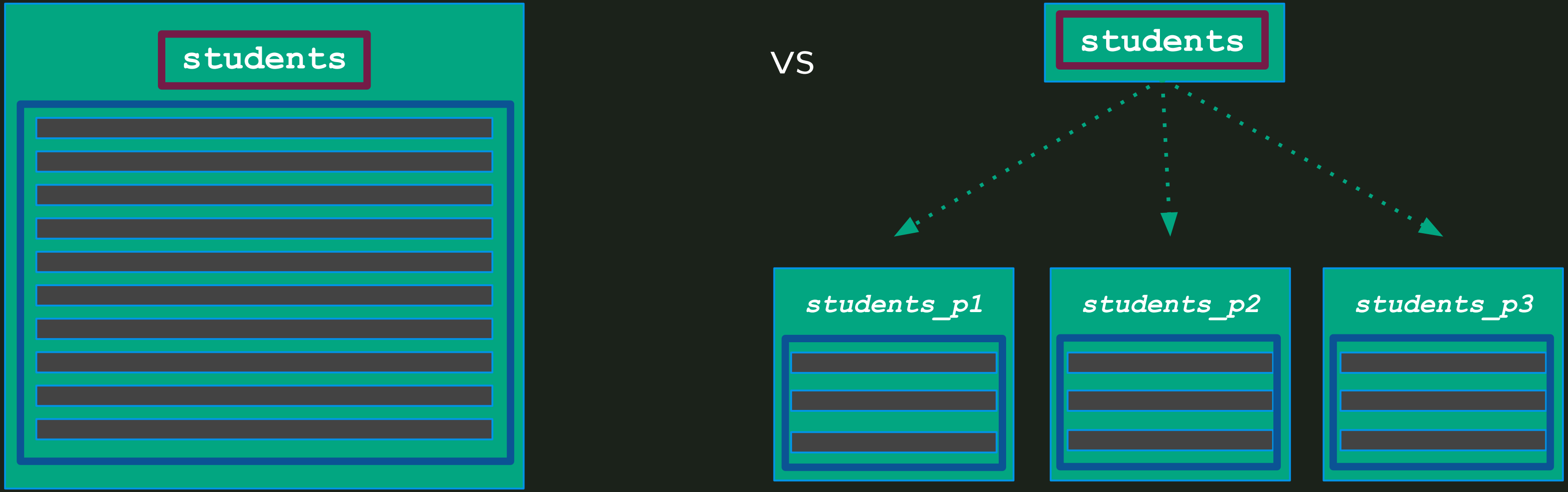
What is partitioning?

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

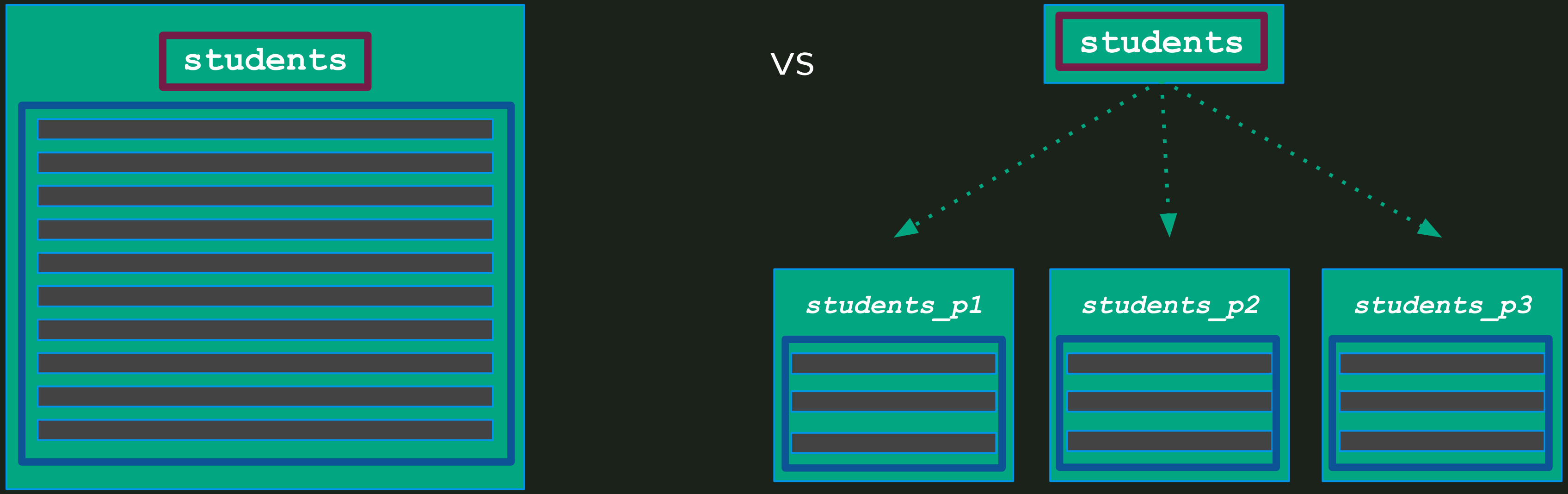


What is partitioning?

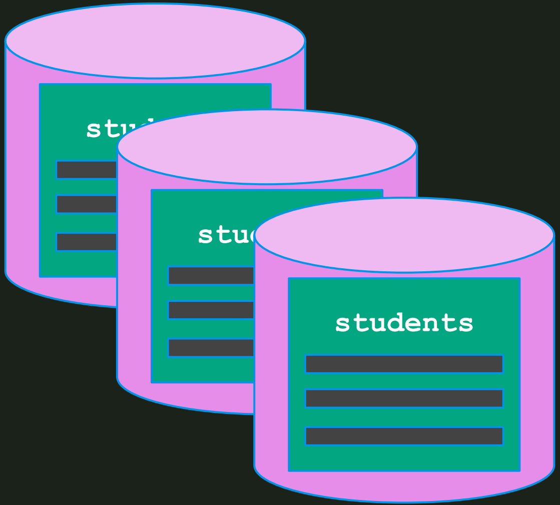
Splitting 1 larger, logical table into n smaller, physical tables [1]

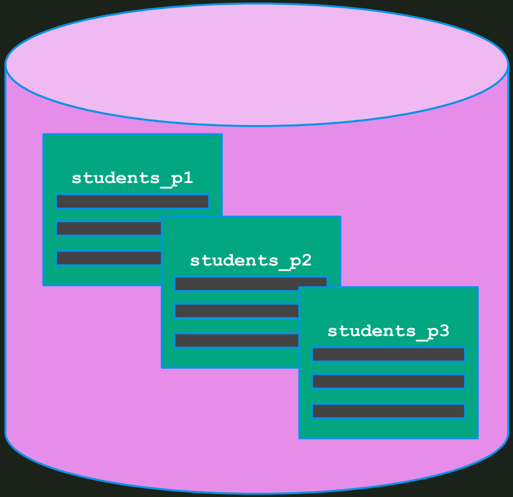


```
SELECT id, full_name FROM students WHERE id = 1;
```



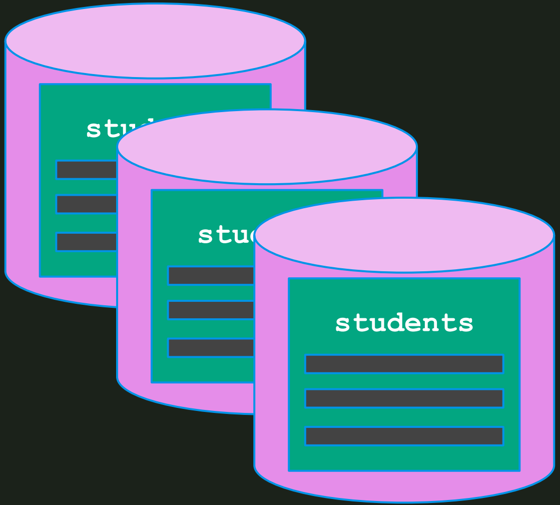
Sharding vs partitioning





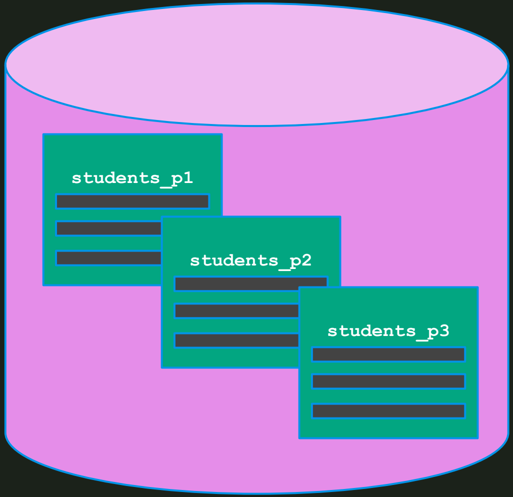
Sharding:

`n` nodes, 1 table/node



Partitioning:

1 node, `n` tables/node



Partitioning in Postgres

- **PG 9.6: partitioning via "table inheritance"**
  - Manual creation, trigger-based `INSERT`s

Difficult setup, bad performance

- **PG 10: declarative partitioning**
  - `CREATE TABLE … PARTITION BY …`
  - `INSERT` "tuple routing", `SELECT` pruning

Easy syntax, basic features

- **PG 11:**
  - Default partition, hash type, `UPDATE` "tuple routing", partition wise `JOIN`, & more

Solid features, broadly usable

Partitioning in Postgres

- **PG 12 – PG18+:**
  - `ATTACH/DETACH` partition concurrently
  - Partition pruning improvements
  - Logical replication for partitioned tables
  - `SPLIT/MERGE` partitions
  - & much more

} Mature, first-class Postgres feature

2. Partitioning methods

1. Range
2. List
3. Hash

Partition key:

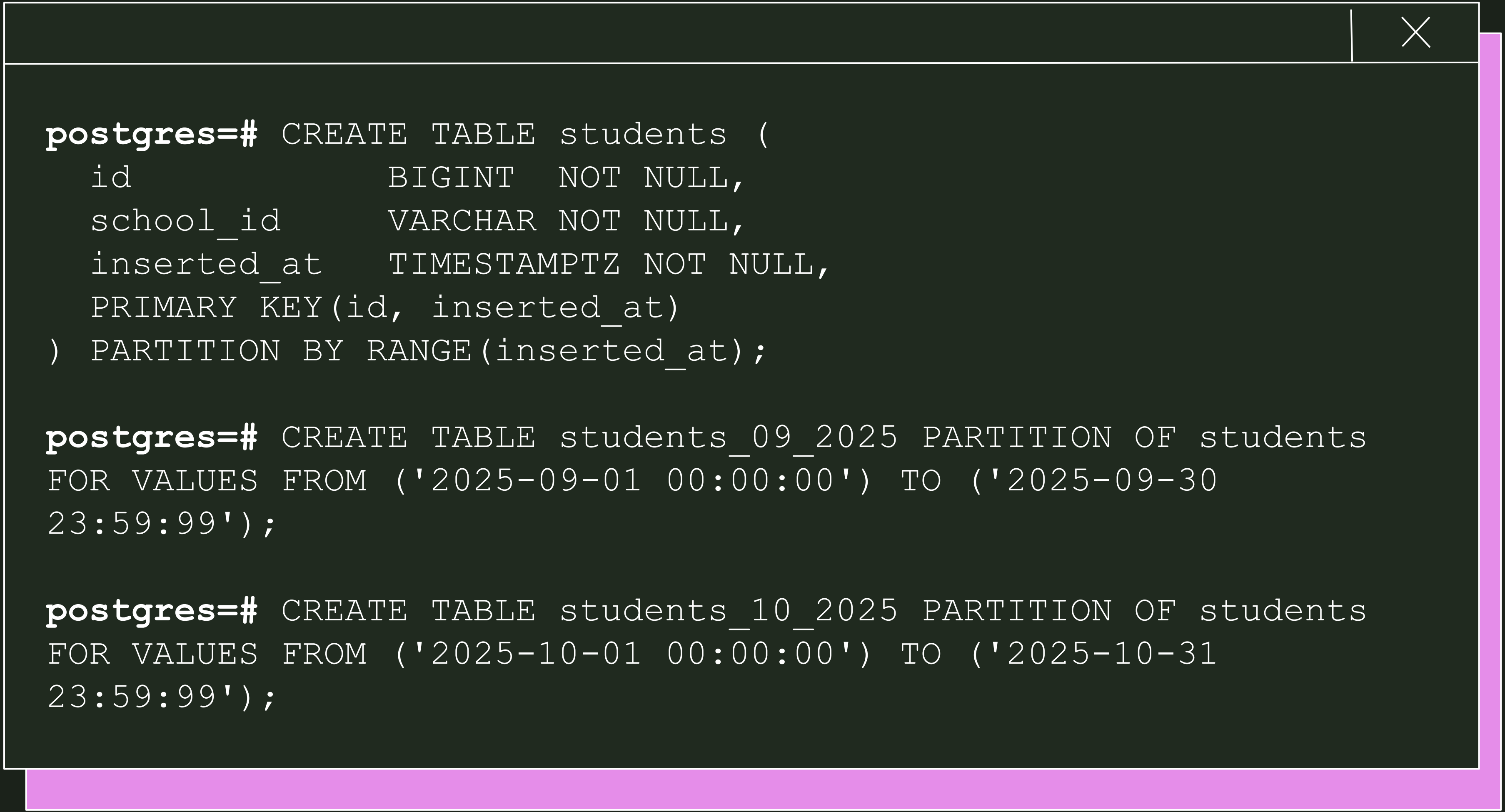
How is data split
into multiple tables?

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



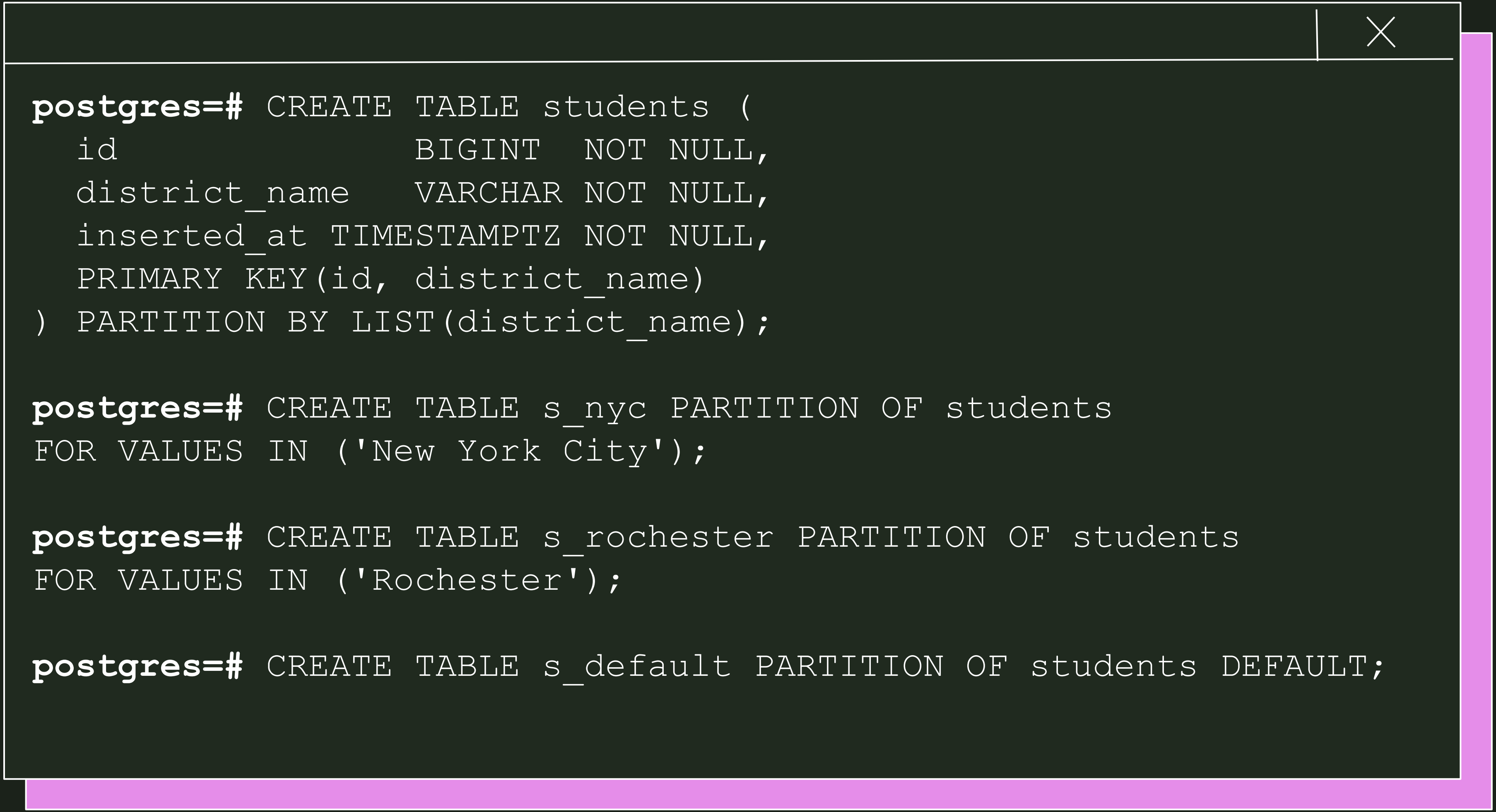
```
postgres=# CREATE TABLE students (
  id            BIGINT  NOT NULL,
  school_id     VARCHAR NOT NULL,
  inserted_at   TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id, inserted_at)
) PARTITION BY RANGE(inserted_at);

postgres=# CREATE TABLE students_09_2025 PARTITION OF students
FOR VALUES FROM ('2025-09-01 00:00:00') TO ('2025-09-30
23:59:99');

postgres=# CREATE TABLE students_10_2025 PARTITION OF students
FOR VALUES FROM ('2025-10-01 00:00:00') TO ('2025-10-31
23:59:99');
```

2. List partitioning

- Partitioning based on explicit column value options
- Low cardinality values, skewed partition table size

Examples:

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



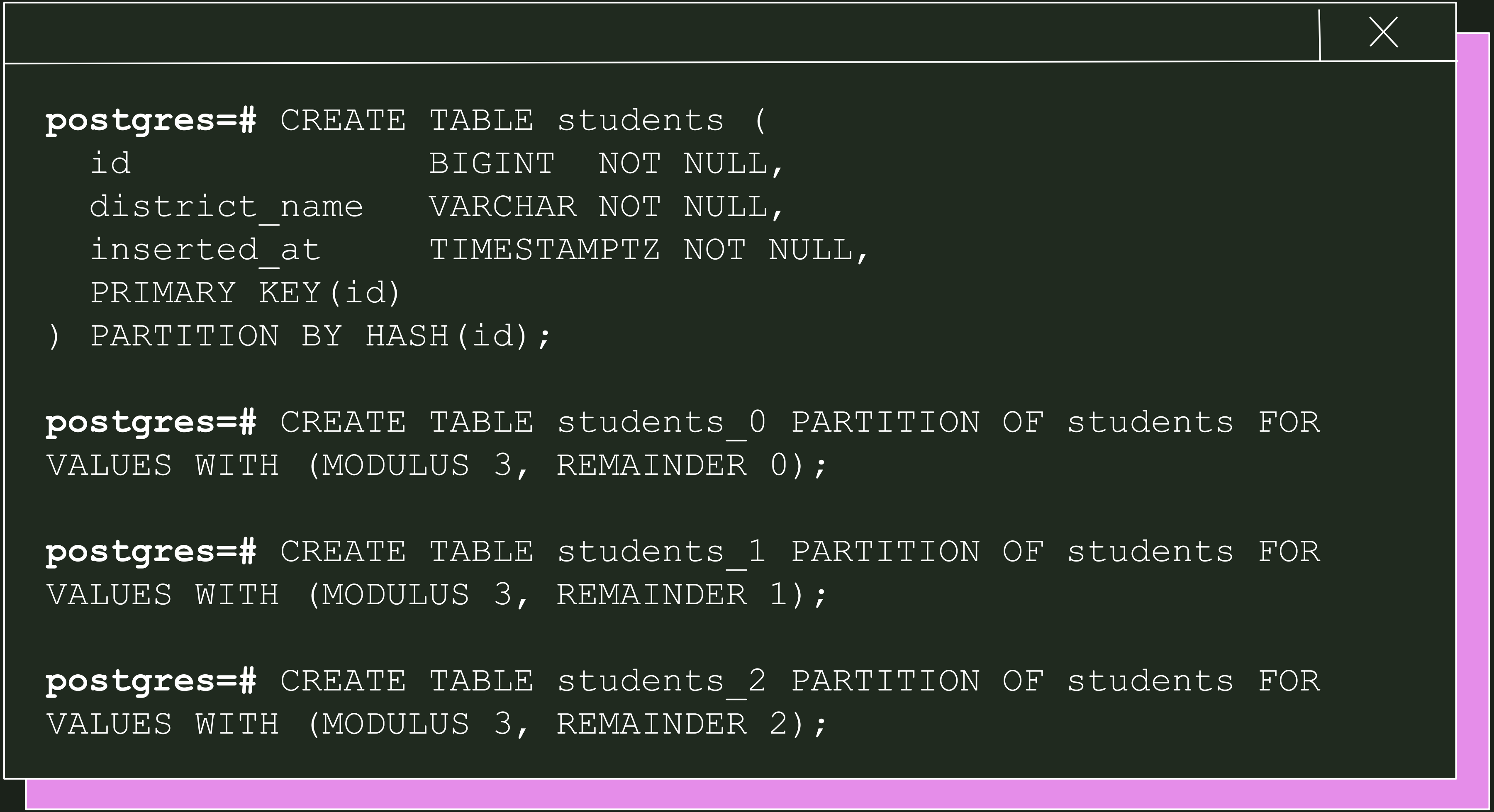
```
postgres=# CREATE TABLE students (
  id                BIGINT  NOT NULL,
  district_name   VARCHAR NOT NULL,
  inserted_at TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id, district_name)
) PARTITION BY LIST(district_name);

postgres=# CREATE TABLE s_nyc PARTITION OF students
FOR VALUES IN ('New York City');

postgres=# CREATE TABLE s_rochester PARTITION OF students
FOR VALUES IN ('Rochester');

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

3. Hash partitioning

- Hashed column value, defining `MODULUS` & `REMAINDER`
- Distributes values evenly

Examples:

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

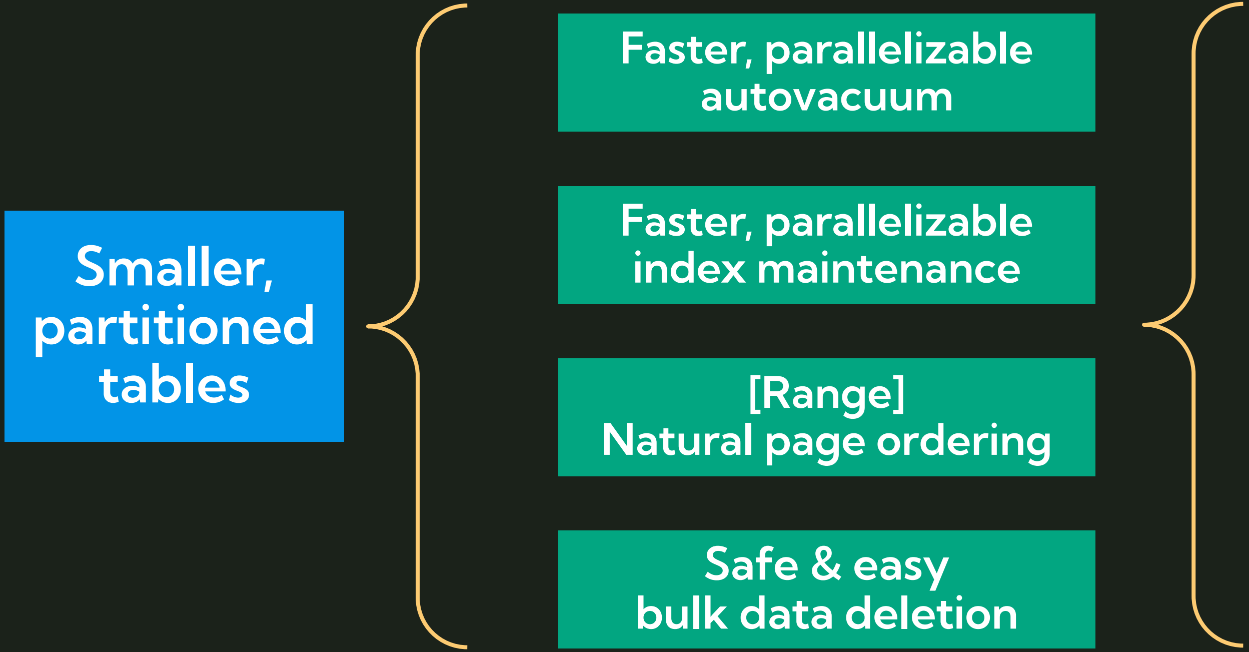


```
postgres=# CREATE TABLE students (
  id              BIGINT  NOT NULL,
  district_name   VARCHAR NOT NULL,
  inserted_at     TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id)
) PARTITION BY HASH(id);

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 impact

Potential impact



Smaller, partitioned tables

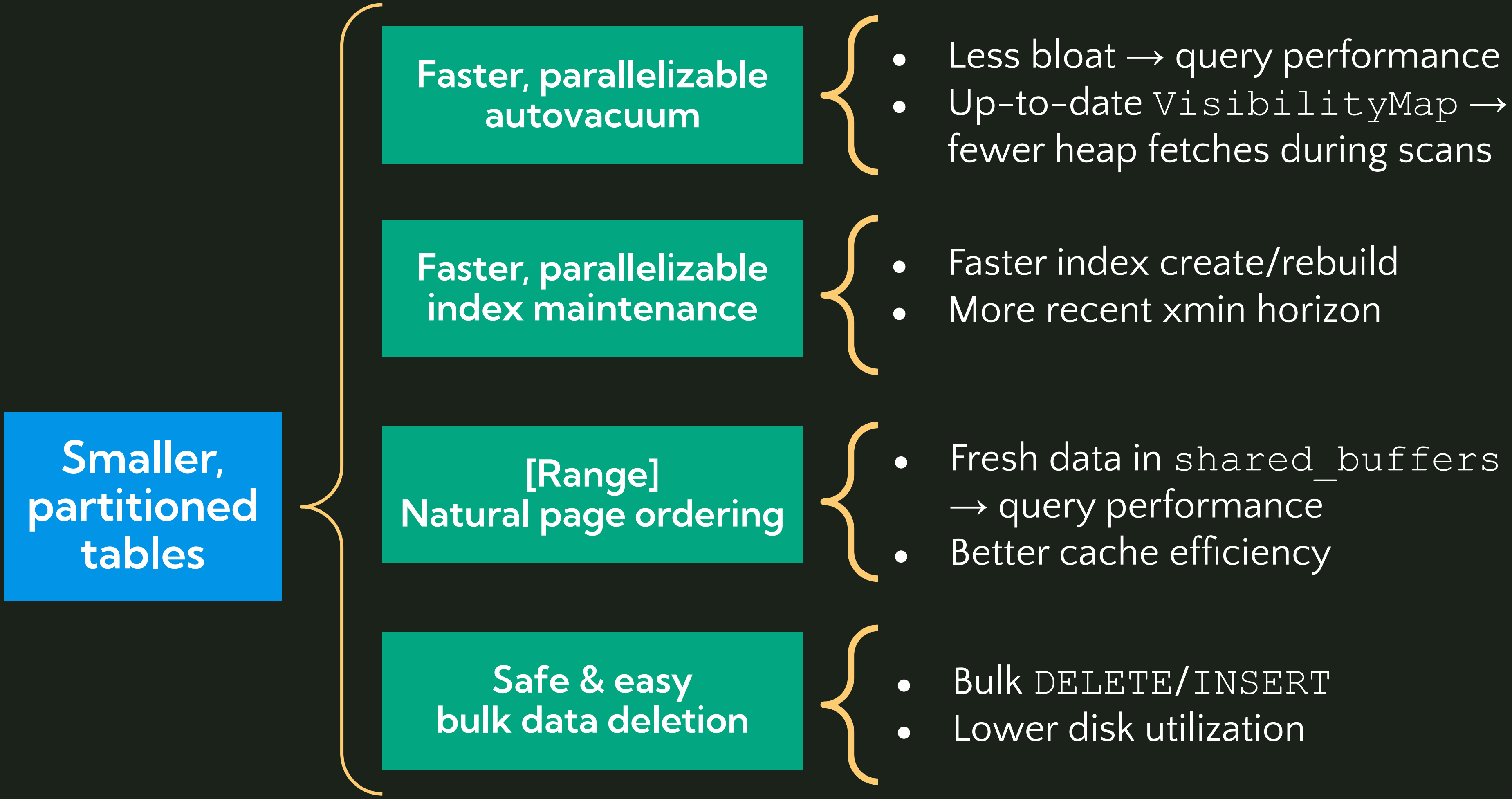
Faster, parallelizable autovacuum

Faster, parallelizable index maintenance

[Range] Natural page ordering

Safe & easy bulk data deletion

- Query performance improvements
- Bloat reduction
- Better cache efficiency



Smaller, partitioned tables

- **Faster, parallelizable autovacuum**
  - Less bloat → query performance
  - Up-to-date `VisibilityMap` → fewer heap fetches during scans
- **Faster, parallelizable index maintenance**
  - Faster index create/rebuild
  - More recent xmin horizon
- **[Range] Natural page ordering**
  - Fresh data in `shared_buffers` → query performance
  - Better cache efficiency
- **Safe & easy bulk data deletion**
  - Bulk `DELETE`/`INSERT`
  - Lower disk utilization

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

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

DENIED

Downsides of partitioning

- Possible negative impact on performance

- Stronger Postgres knowledge required from app developers

- Advanced features → advanced expertise
  - Knowledge of “gotchas”

When is partitioning “worth it”?

Industry rule-of-thumb

- Table size >=100GB (at least) ⭐

Postgres docs

- Table size > physical memory of the server

👍 My rules-of-thumb

`RANGE` partitioning

- Typically the best ROI
- If you have a “natural” range partition key or want to “expire” old data

`LIST` partitioning

- If you need to regularly bulk `DELETE` or `INSERT` data for a group

`HASH` partitioning

- Partitioning is needed for maintenance reasons, but no natural PK
- No plans to “expire” partitions

Downsides of partitioning

- Possible negative impact on performance
- Stronger Postgres knowledge required from app developers
- Advanced features → advanced expertise
  - **Knowledge of “gotchas”**

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,
  school_id     VARCHAR NOT NULL,
  inserted_at   TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id, inserted_at)
) PARTITION BY RANGE(inserted_at);
```

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

Migrate PRIMARY KEY to a composite key

- Beware of `UPSERT`s
- `id` no longer UNIQUE

🔥 Rapid Fire Gotchas

- `DEFAULT` partition
- `HASH` partitioning
  - Range queries (i.e., `WHERE <partition_key> BETWEEN x, y`) can't use partition pruning
  - Partition count cannot be changed
- Logical replication: `publish_via_partition_root`

4. Partitioning an existing table

Why is this a challenge?

- Tables are typically partitioned retroactively
- No support for “`ALTER TABLE … PARTITION BY`”

‼️ Disclaimer

There are MANY ways to partition tables. This talk focuses on native Postgres, not extensions.

- pg_partman
- pgslice
- pg_party
- pglogical

Extensions which provide utilities relevant to partitioning methods

🏫 Case Study: NY Dept of Education

- Table size (GB)
- Query patterns
  - Read vs write
  - Bulk load/delete
  - Filters
- Maintenance window length
- Disk availability
- Budget

Use Case #1: Offline migration

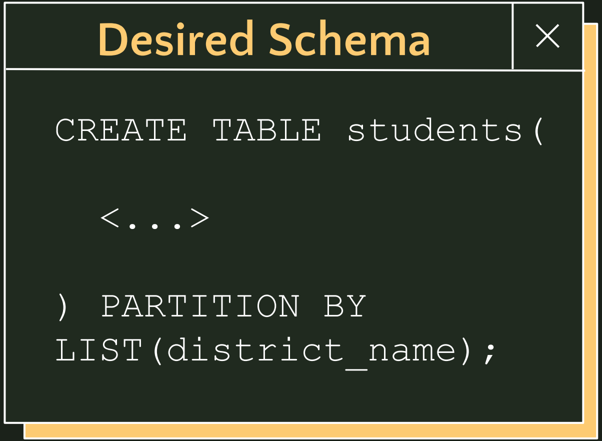
180GB table

- ○ 90% reads
- ○ 10% writes

- Frequent bulk load/delete by `district_name`
- Traffic during school hours
- Low DBA budget (teachers paid well)

Constraints:

- ✅ <=3 hours maintenance window
- ✅ 300GB disk space available



Desired Schema

```
CREATE TABLE students(

  <...>

) PARTITION BY
LIST(district_name);
```

```
-- Step #1: Create a LIST partitioned table & partitions.


postgres=# CREATE TABLE students_v2 (
  id              BIGINT  NOT NULL,
  district_name   VARCHAR NOT NULL,
  inserted_at TIMESTAMPTZ NOT NULL,
  PRIMARY KEY(id, district_name)
) PARTITION BY LIST(district_name);

postgres=# CREATE TABLE s_nyc PARTITION OF students_v2
FOR VALUES IN ('New York City');

<...>

postgres=# CREATE INDEX students__district_name ON students_v2
(district_name);
```

```
-- Step #2: Manually insert the data
 -  - INSERT (example below), single or batched
 -  - pg_partman[1]
 -  - pg_dump/load



postgres=# BEGIN;

INSERT INTO students_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


ALTER TABLE students RENAME TO students_old;
ALTER TABLE students_v2 RENAME TO students;


postgres=# COMMIT;


-- Step #4: Drop “students_old”

postgres=# DROP TABLE students_old;
```

Use Case #2: Online migration, duplicating tables

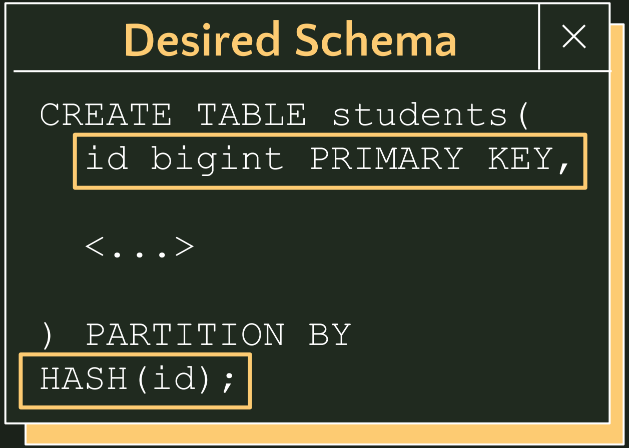
400GB table

- ○ 60% reads
- ○ 40% writes

- Traffic distributed roughly 24/7
- District has issues with maintenance runtime
- 2x data growth expected this year, and query patterns/filters are variable

Constraints:

- ⚠️ <=3m downtime acceptable
- ✅ 600GB disk space available



Desired Schema

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

   <...>

) PARTITION BY
HASH(id);
```

```
-- Step #1: Create a HASH partitioned table & partitions.


postgres=# CREATE TABLE students_v2 (
    LIKE students
    INCLUDING DEFAULTS INCLUDING INDEXES INCLUDING CONSTRAINTS
) PARTITION BY HASH(id);

postgres=# CREATE TABLE s_0 PARTITION OF students_v2 FOR VALUES
WITH (MODULUS 10, REMAINDER 0);

<...>

postgres=# CREATE TABLE s_9 PARTITION OF students_v2 FOR VALUES
WITH (MODULUS 10, REMAINDER 9);
```

```
-- Step #2: Create a function returning a trigger to duplicate
incoming INSERT/UPDATE/DELETE/MERGE operations to students_v2


postgres=# CREATE OR REPLACE FUNCTION duplicate_dml()

    RETURNS TRIGGER AS
    $$
    BEGIN
        <...>
    END;
    $$ LANGUAGE PLPGSQL;
```

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

```
–- Step #3: Create a trigger, so the function is called after
INSERT/UPDATE/DELETE/MERGE on the “students” table.



postgres=#

CREATE TRIGGER duplicate_dml_trigger
       AFTER INSERT OR UPDATE OR DELETE ON students
       FOR EACH ROW EXECUTE PROCEDURE
partition_migrate();
```

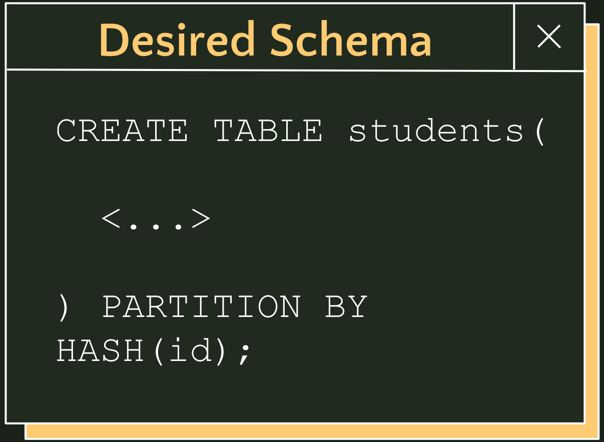
```
-- Step #4: Copy all data from “students” to “students_v2” in
batches. On PK conflict, do nothing.

-- Step #5: Once backfill is complete, “swap” the two tables & drop
the old table.


postgres=#

BEGIN;
    ALTER TABLE students RENAME TO students_old;
    ALTER TABLE student_v2 RENAME TO students;
COMMIT;

postgres=# DROP TABLE students_archived;
```

Use Case #3: Online migration, no table duplication

400GB table

- ○ 60% reads
- ○ 40% writes

- Traffic distributed roughly 24/7
- District has issues with maintenance runtime
- 2x data growth expected this year, and query patterns/filters are variable



Desired Schema

```
CREATE TABLE students(

   <...>

) PARTITION BY
HASH(id);
```

Constraints:

- ⚠️ <=3m maintenance window
- ⚠️ 100GB disk space available

Doesn’t have 2x disk space

```
-- Step #1: Create a HASH partitioned table & partitions.


postgres=# CREATE TABLE students_v2 (
    LIKE students
    INCLUDING DEFAULTS INCLUDING INDEXES INCLUDING CONSTRAINTS
) PARTITION BY HASH(id);

postgres=# CREATE TABLE s_0 PARTITION OF students_v2 FOR VALUES WITH
(MODULUS 10, REMAINDER 0);

<...>

postgres=# CREATE TABLE s_9 PARTITION OF students_v2 FOR VALUES WITH
(MODULUS 10, REMAINDER 9);
```

```
-- Step #2: Create a function returning a trigger:
      - ON INSERT: insert only to new table
      - ON DELETE: delete from both new & old table
      - ON UPDATE: delete from old table, upsert to new table


postgres=# CREATE OR REPLACE FUNCTION partition_migrate()

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 “students” with a UNION view of both tables. Create a
trigger which calls partition_migrate() in lieu of INSERT/UPDATE/DELETE.

postgres=# BEGIN;

    ALTER TABLE students RENAME TO students_old;

    CREATE VIEW students AS
        SELECT id, <data> FROM students_old
        UNION ALL
        SELECT id, <data> FROM students_v2
    ;

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

COMMIT;
```

```
-- Step #4: Copy all data from “students” to “students_v2” in batches


-- Step #5: Drop the view and migration function. Rename the new,
partitioned table to be “students”. Drop “students_old”.

postgres=#

BEGIN;
    DROP VIEW students;
    DROP FUNCTION partition_migrate();
    ALTER TABLE students_v2 RENAME TO students;
COMMIT;

postgres=# DROP TABLE students_old;
```

Use Case #4: Logical replication 🚀📈

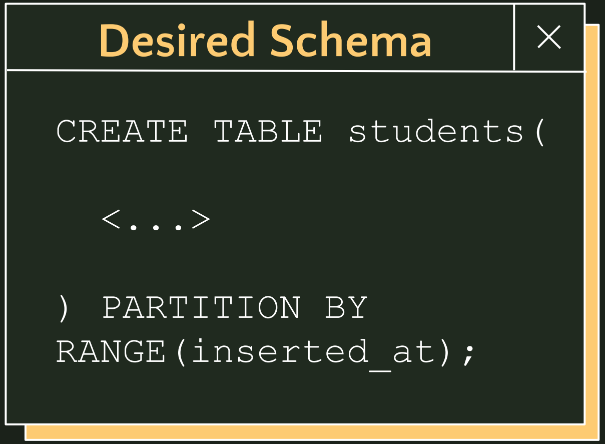
4TB table

- 80% reads
- 20% writes

- Traffic distributed roughly 24/7
- Most queries filter by `grad_date`
- High DBA budget, and partitioning process must be repeatable

Constraints:

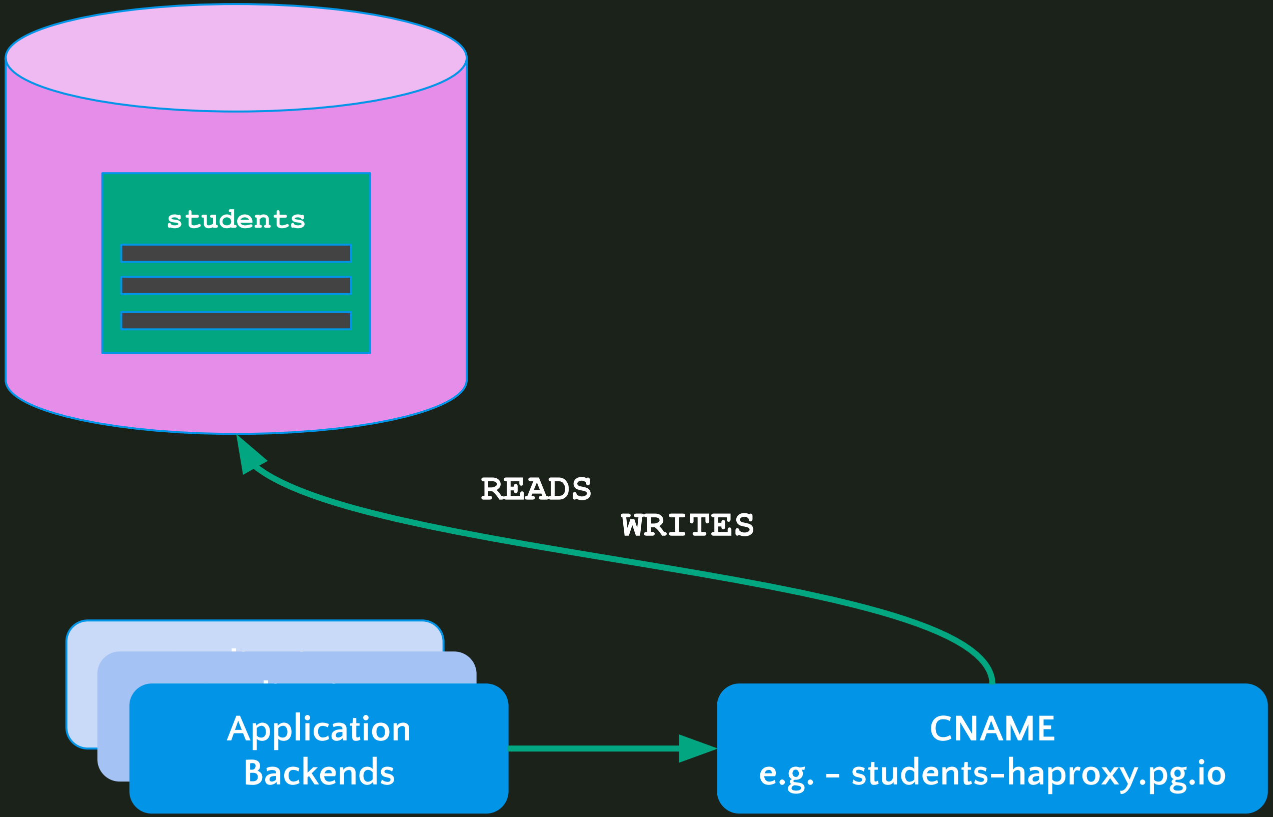
- ⚠️ <=1m maintenance window
- ⚠️ 300GB disk space available
- ⚠️ Task must be easily repeatable



Desired Schema

```
CREATE TABLE students(

  <...>

) PARTITION BY
RANGE(inserted_at);
```

Use Case #4: Logical replication 🚀📈



students

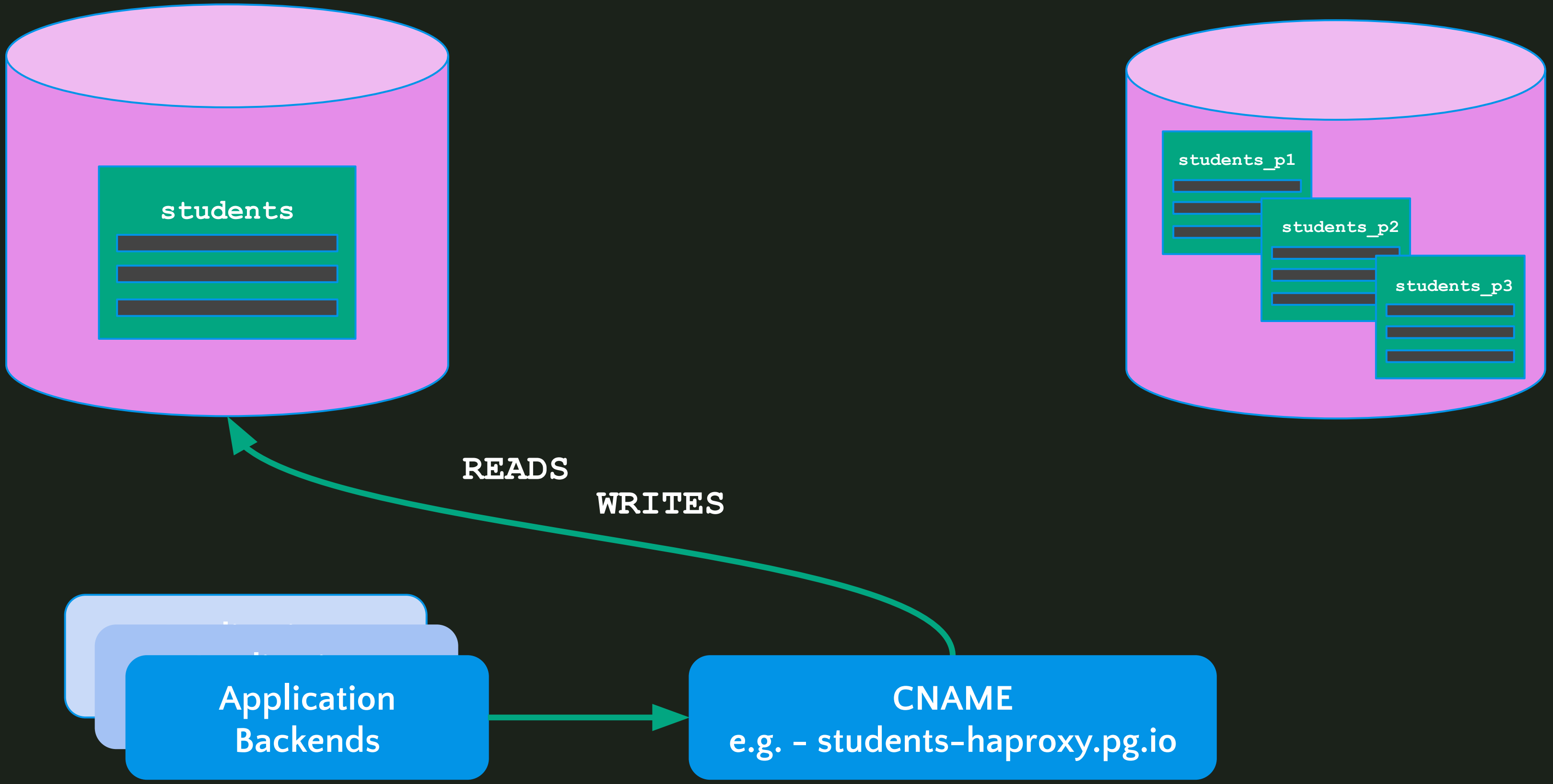
READS

WRITES

Application Backends

CNAME
e.g. - students-haproxy.pg.io

Use Case #4: Logical replication 🚀📈



students

students_p1

students_p2

students_p3

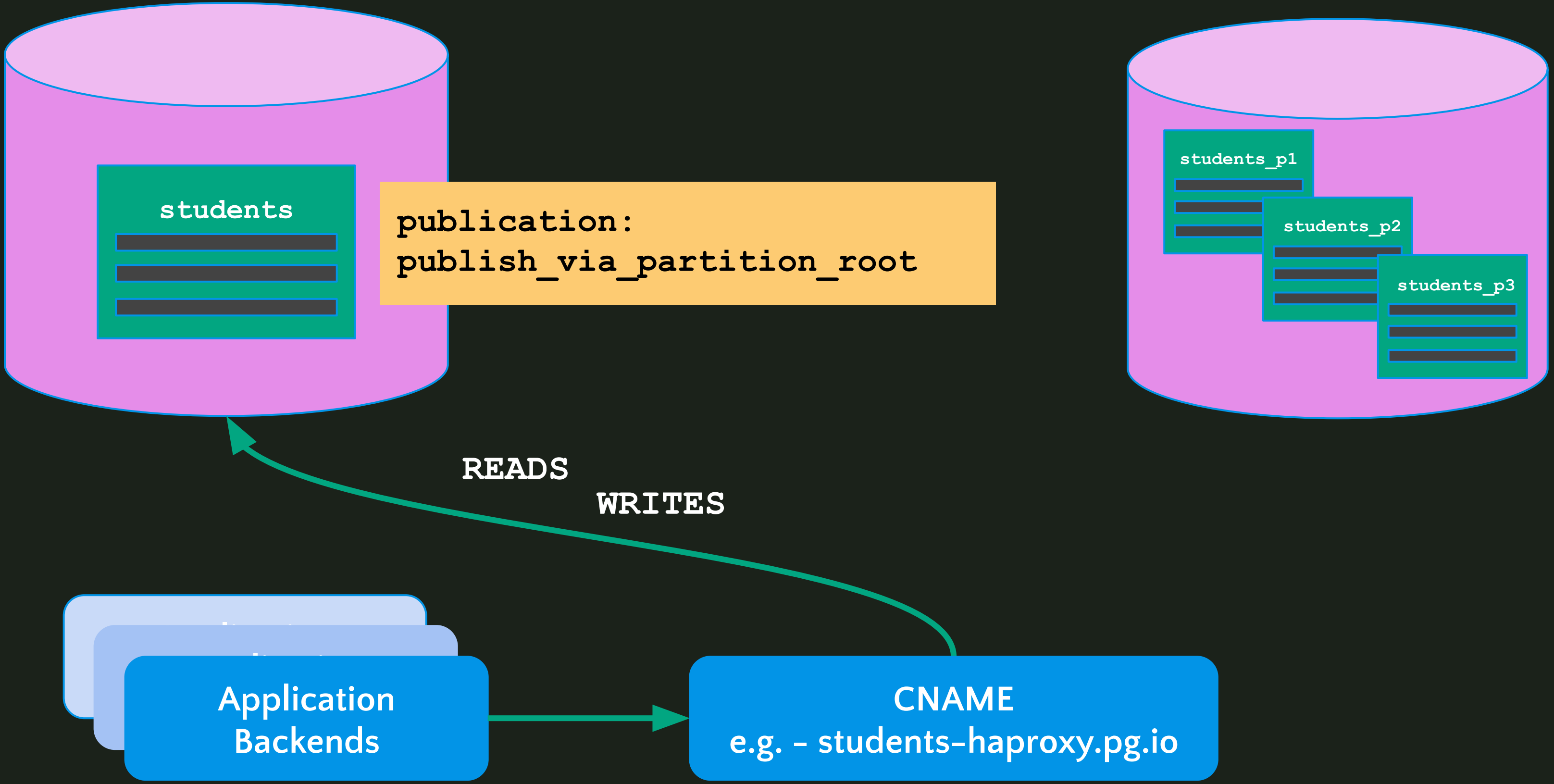
READS

WRITES

Application
Backends

CNAME
e.g. - students-haproxy.pg.io

Use Case #4: Logical replication 🚀📈



students

publication:
publish_via_partition_root

students_p1

students_p2

students_p3

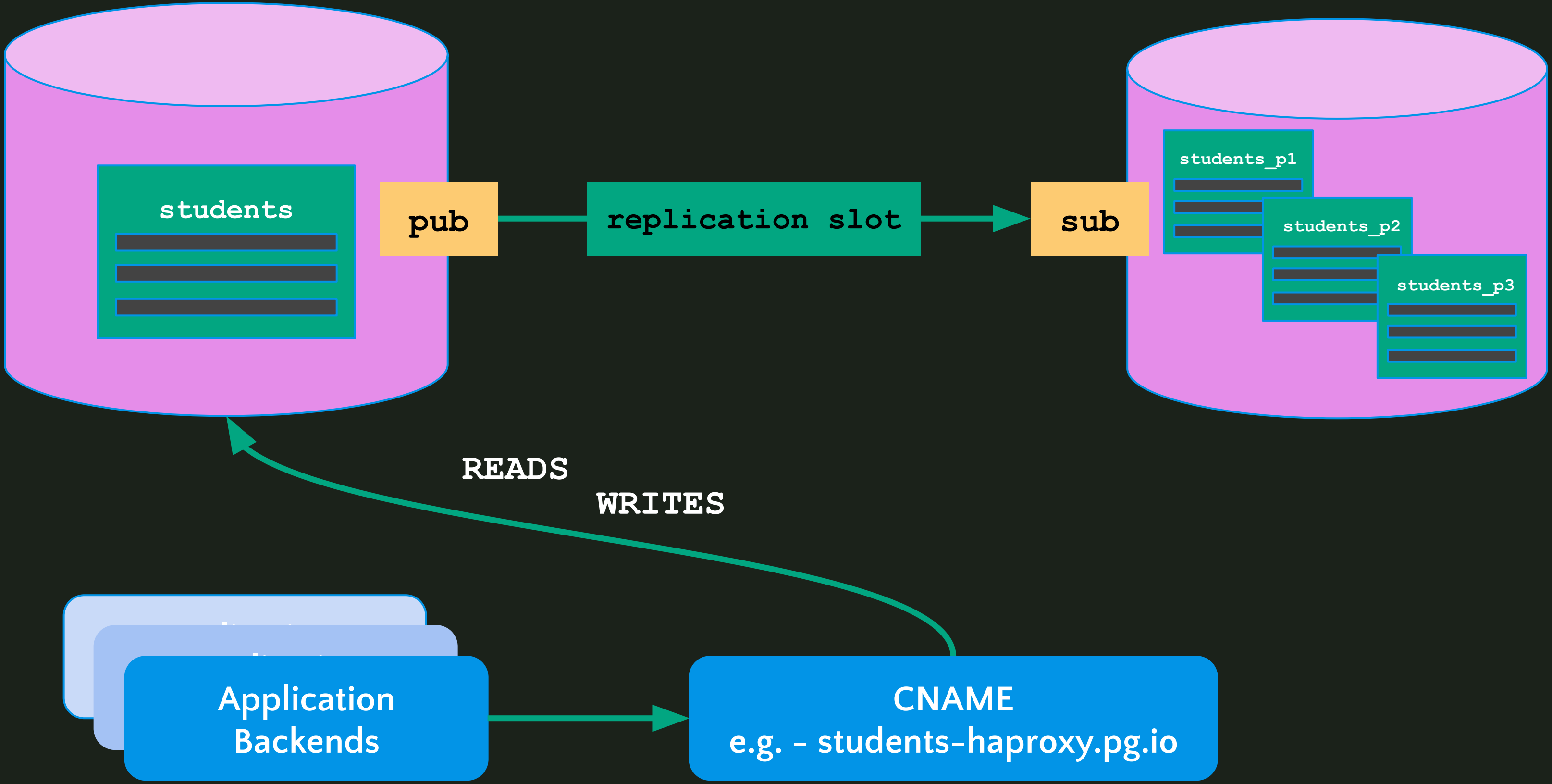
READS

WRITES

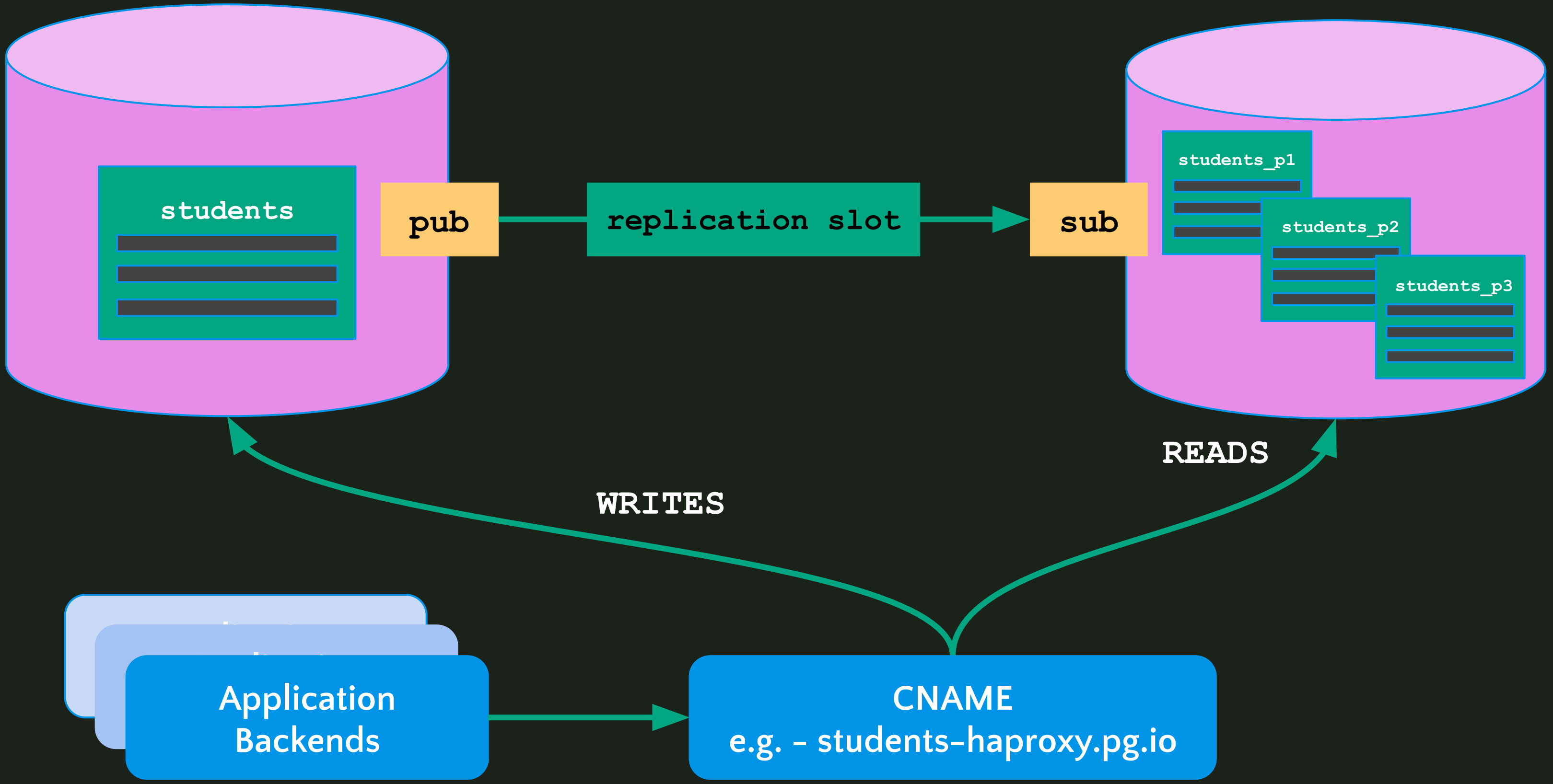
Application
Backends

CNAME
e.g. - students-haproxy.pg.io

Use Case #4: Logical replication 🚀📈



Use Case #4: Logical replication 🚀📈



students

pub

replication slot

sub

students_p1

students_p2

students_p3

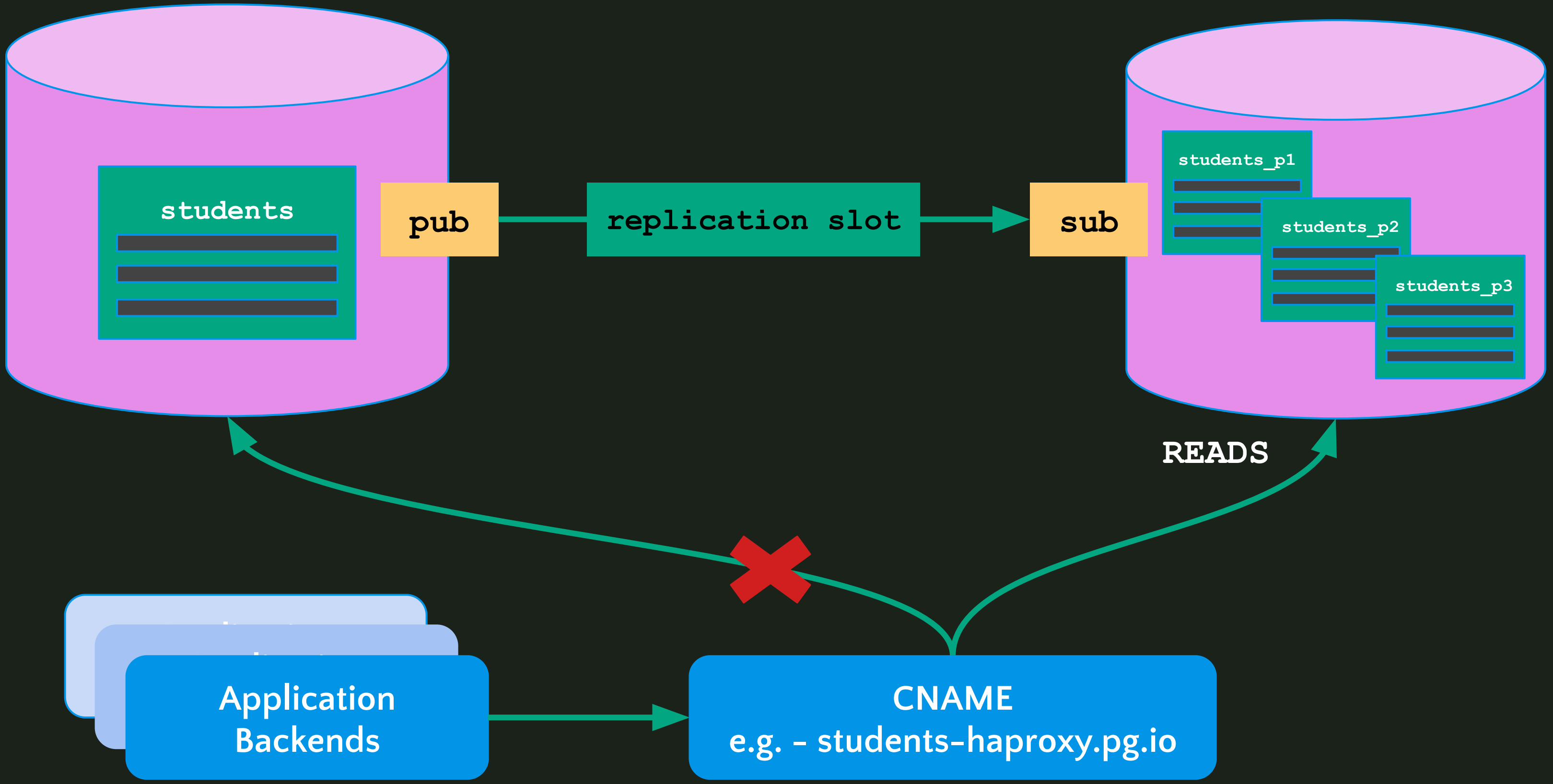
READS

WRITES

Application Backends

CNAME
e.g. - students-haproxy.pg.io

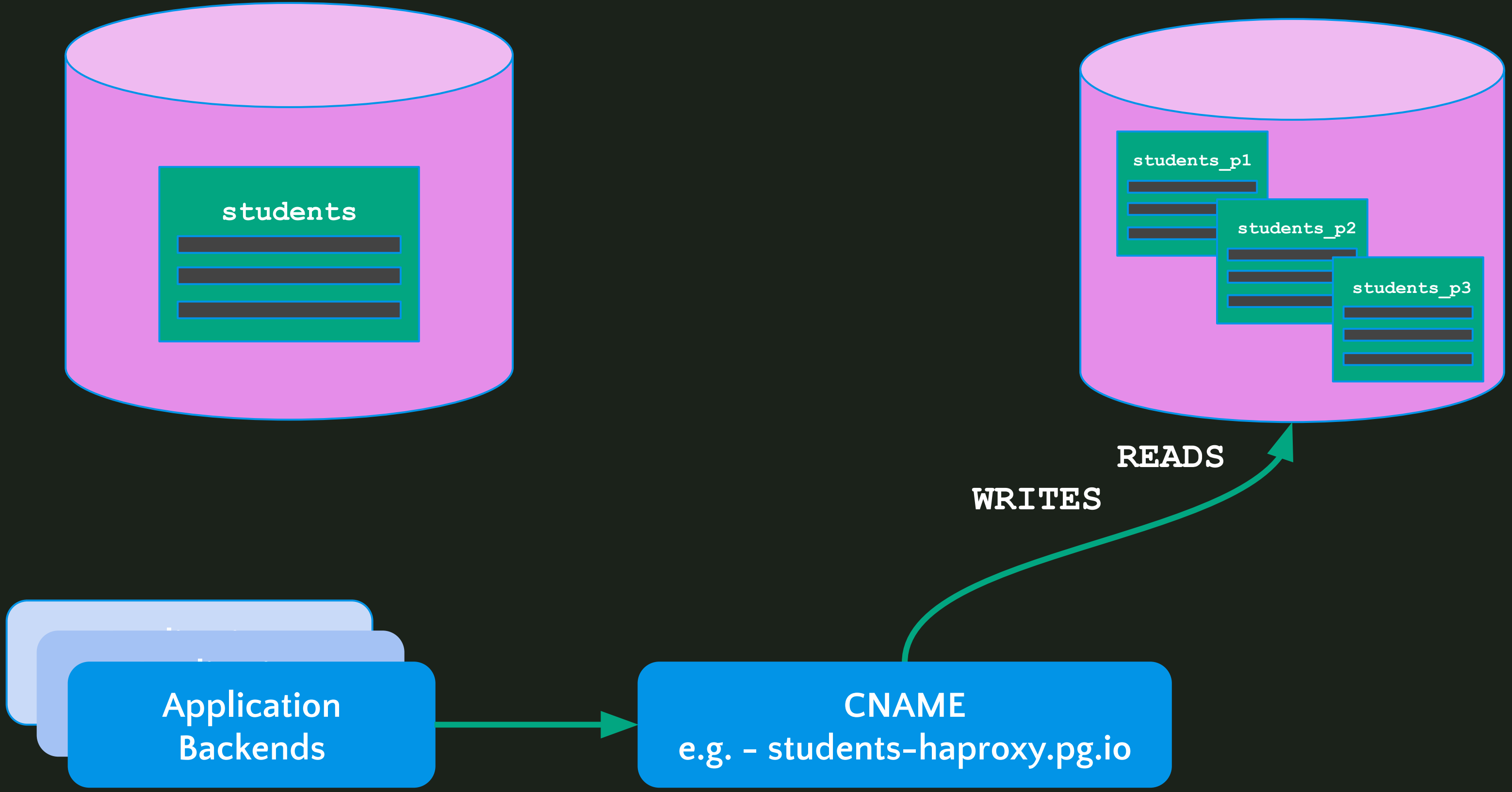
Use Case #4: Logical replication 🚀📈



Use Case #4: Logical replication 🚀📈

```
SELECT application_name, pg_current_wal_lsn(),
replay_lsn, pg_wal_lsn_diff(pg_current_wal_lsn(),
replay_lsn)::bigint FROM pg_stat_replication;
```

Use Case #4: Logical replication 🚀📈



Use Case #4: Logical replication 🚀📈

Pre-Checks

- Primary key, large object (`lo`), unlogged tables, etc
- Destination table partitioned

Logical replication

- Publication (`publish_via_partition_root`) & subscription
- No schema changes/DDL

During write downtime

- Sync `SEQUENCES`, refresh `MATERIALIZED VIEWs`
- Disable subscription
- Verify LSN convergence
- CNAME/config propagation

4. Maintenance, Configuration, & Observability

Maintenance

- Regular creation of new partitions

`pg_partman`:

Automatically creates time/number-based partition sets, or detach/delete old partitions

- `CALL partman.run_maintenance_proc(<...>);`

Observability

Monitoring/alerting:

- Partitions are created/deleted by `pg_partman` as expected
- Partition size (skew)

`auto_explain`:

- Dynamically help detect slow query plans

Configuration

Partitioned tables are still just “tables”

`autovacuum_max_workers` (default=3)

- Consider increasing, based on on resource usage

Organizational Support

Build an understanding of partitioning & its benefits/constraints

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!

chelseadole@gmail.com
chelseadole.com