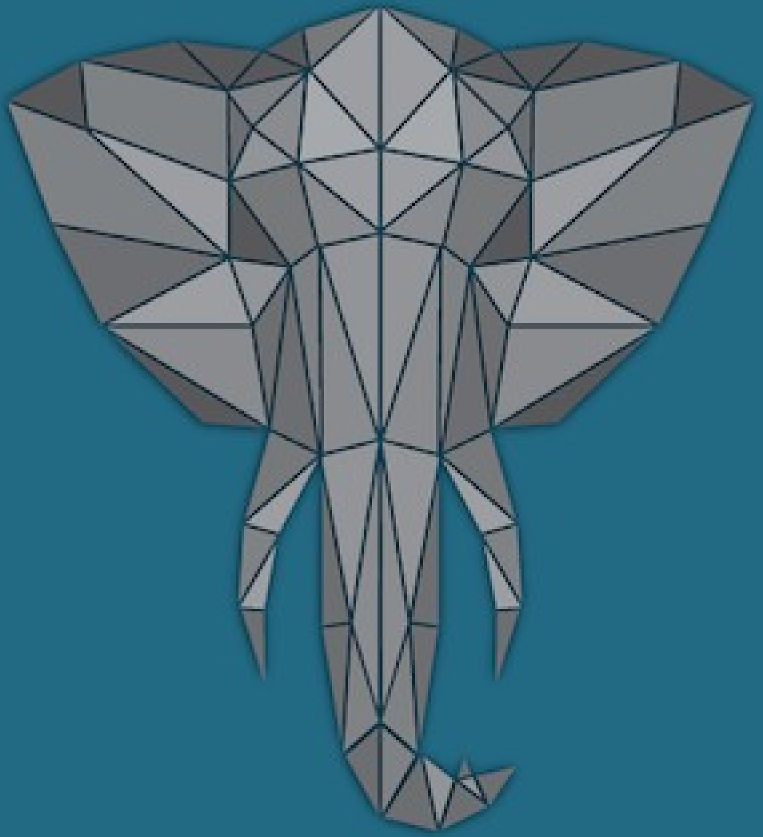
Adventures on live partitioning



ONGRES

Matteo Melli

DEV

Working @ OnGres (www.ongres.com)
Software Developer
PostgreSQL support

@teoincontatto

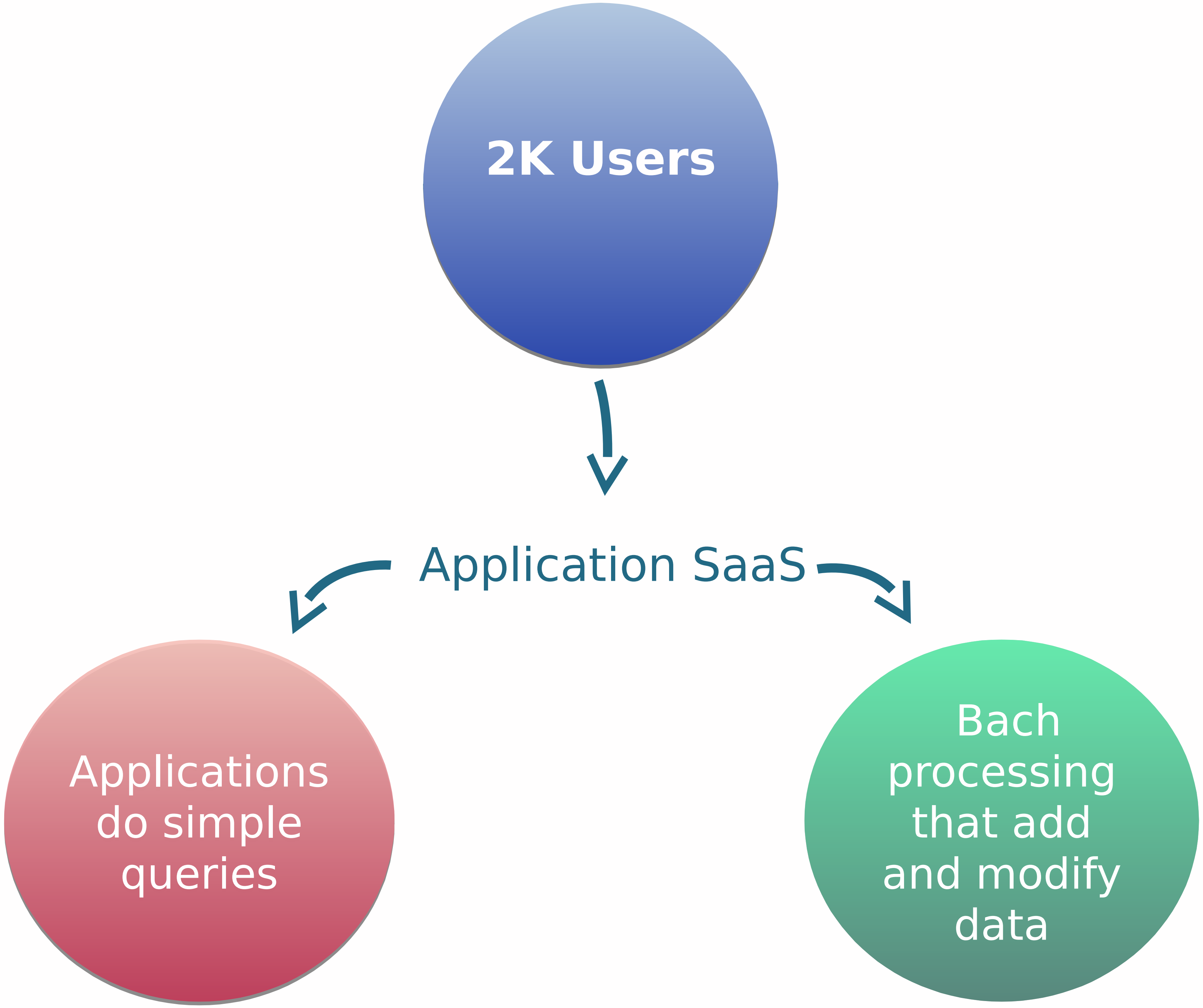
teoincontatto

About ONGRES

- IT firm specialized on R&D on Databases, more specifically PostgreSQL:
  - Training
  - Consulting and development
  - PostgreSQL Support

- Developers of ToroDB (www.torodb.com), an open-source, document-store database that works on top of PostgreSQL and is compatible with MongoDB.
- Partners of www.pythian.com, reference multinational company providing database support and data services.

Our customer



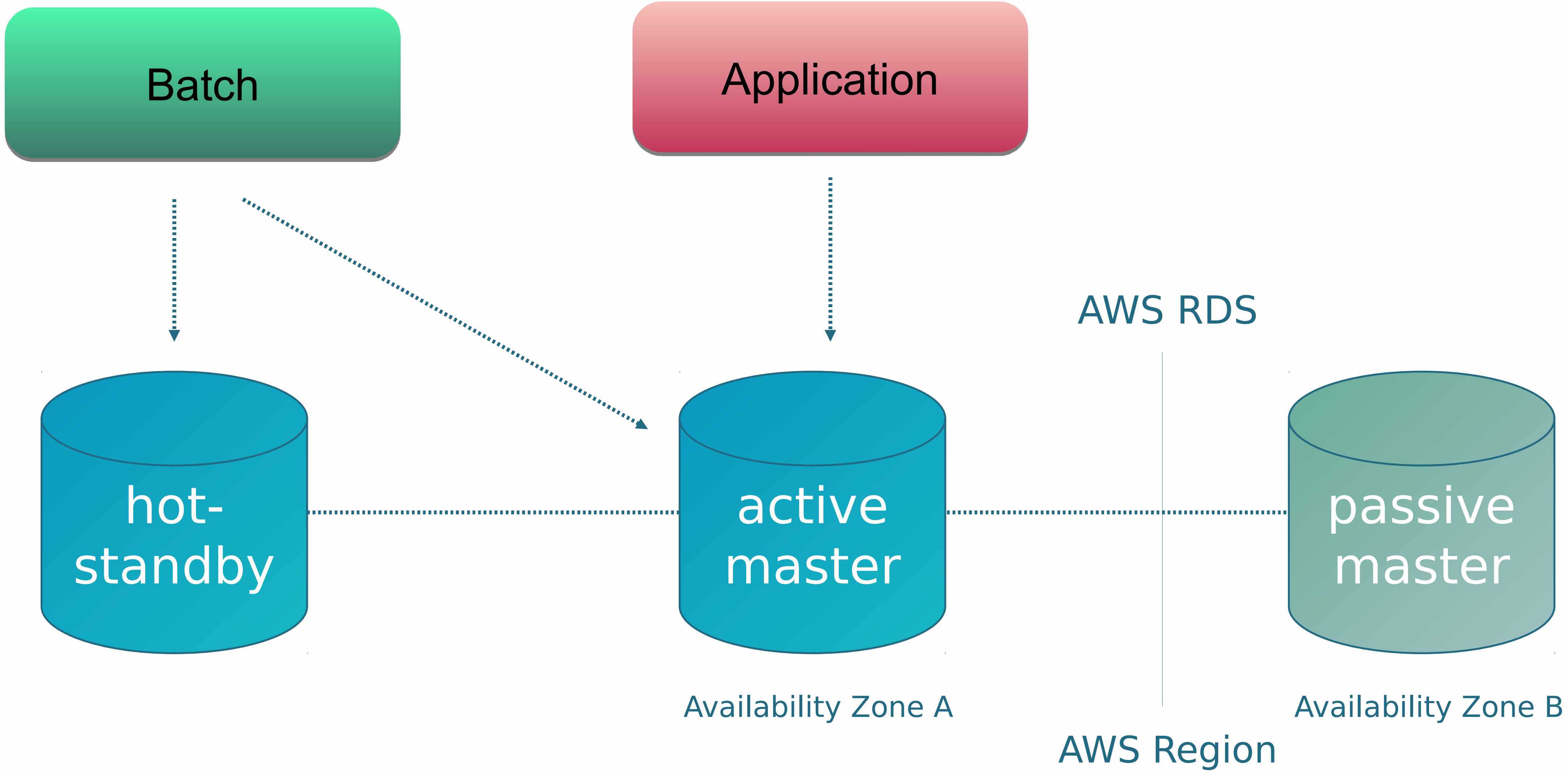
2K Users

Application SaaS

Applications do simple queries

Bach processing that add and modify data

System characteristics



System characteristics

- PostgreSQL 9.6
- Amazon RDS db.r3.xlarge
- 200MB/s throughput (max 1.2GB/s)
- HA with Multi-AZ active/passive
- Hot-standby replica

Big table problem

- ✓ ~100GB size (50% indexes) table
- ✓ ~**1000M** rows table
- ✓ Table growth due to batch process (from few MB to some GB per night)
- ✓ Queries slow down (up to x10 slower)
- ✓ CUD slow down (up to x100 slower)

Big table problem

The requirements

✓ Removing indexes not an option

1

✓ Partitioning is the way to go

2

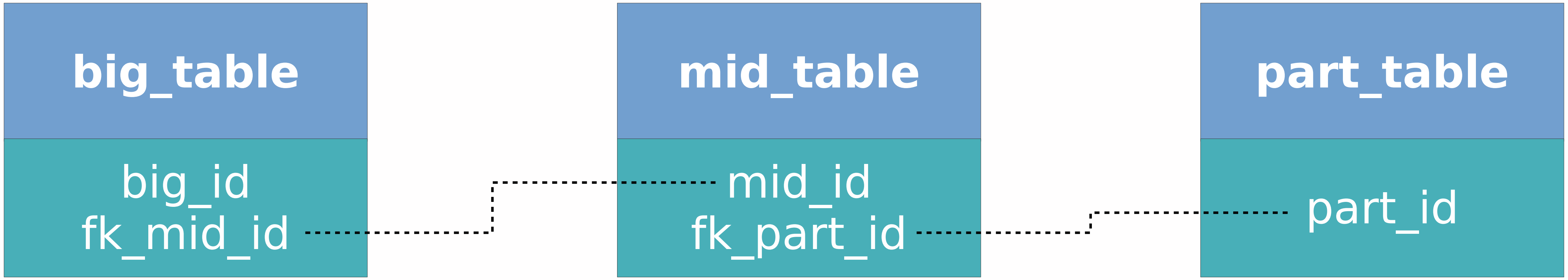
✓ 2 days of outage at maximum

3

Type of partitioning

- Using table INHERITANCE
- Cloning PK and FKs

- Range checks on integer column part_id
- Partition key on a 2 JOINs far away table
  - If fk_mid_id is NULL or fk_part_id is NULL row is logically deleted



big_table

big_id
fk_mid_id

mid_table

mid_id
fk_part_id

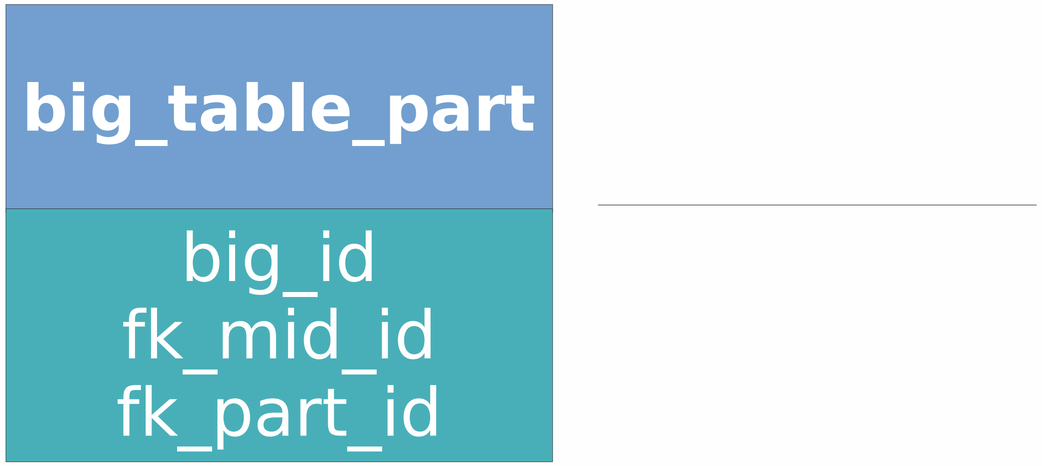
part_table

part_id

The naive approach

- ✓ 2 days maintenance window (will be enough!) 1
- ✓ big_table empty copy plus partition key 2
- ✓ Homogeneous ranges of partition key 3
- ✓ Copy directly to partitions from ad-hoc view 4

The naive approach

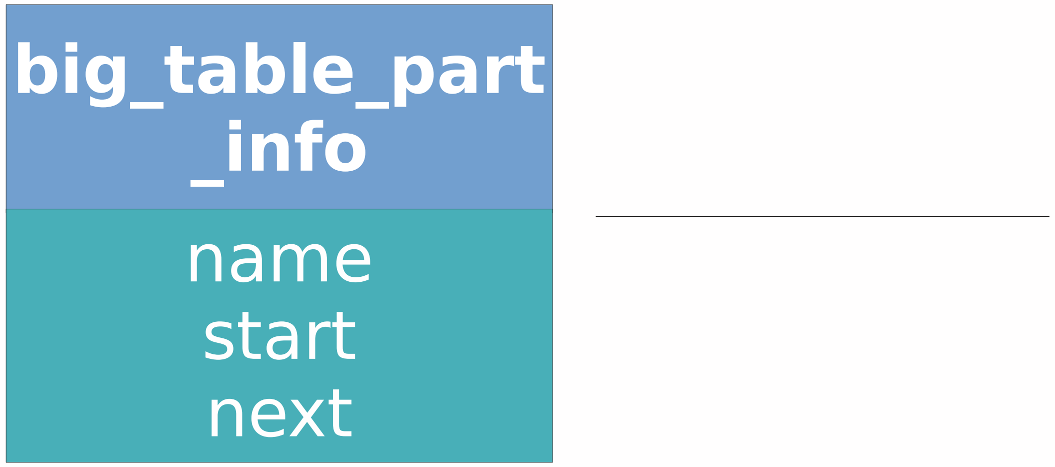


big_table_part

big_id
fk_mid_id
fk_part_id

```
CREATE TABLE big_table_part
(fk_part_id integer)
INHERITS (big_table);
ALTER TABLE big_table_part
NO INHERIT big_table;
```

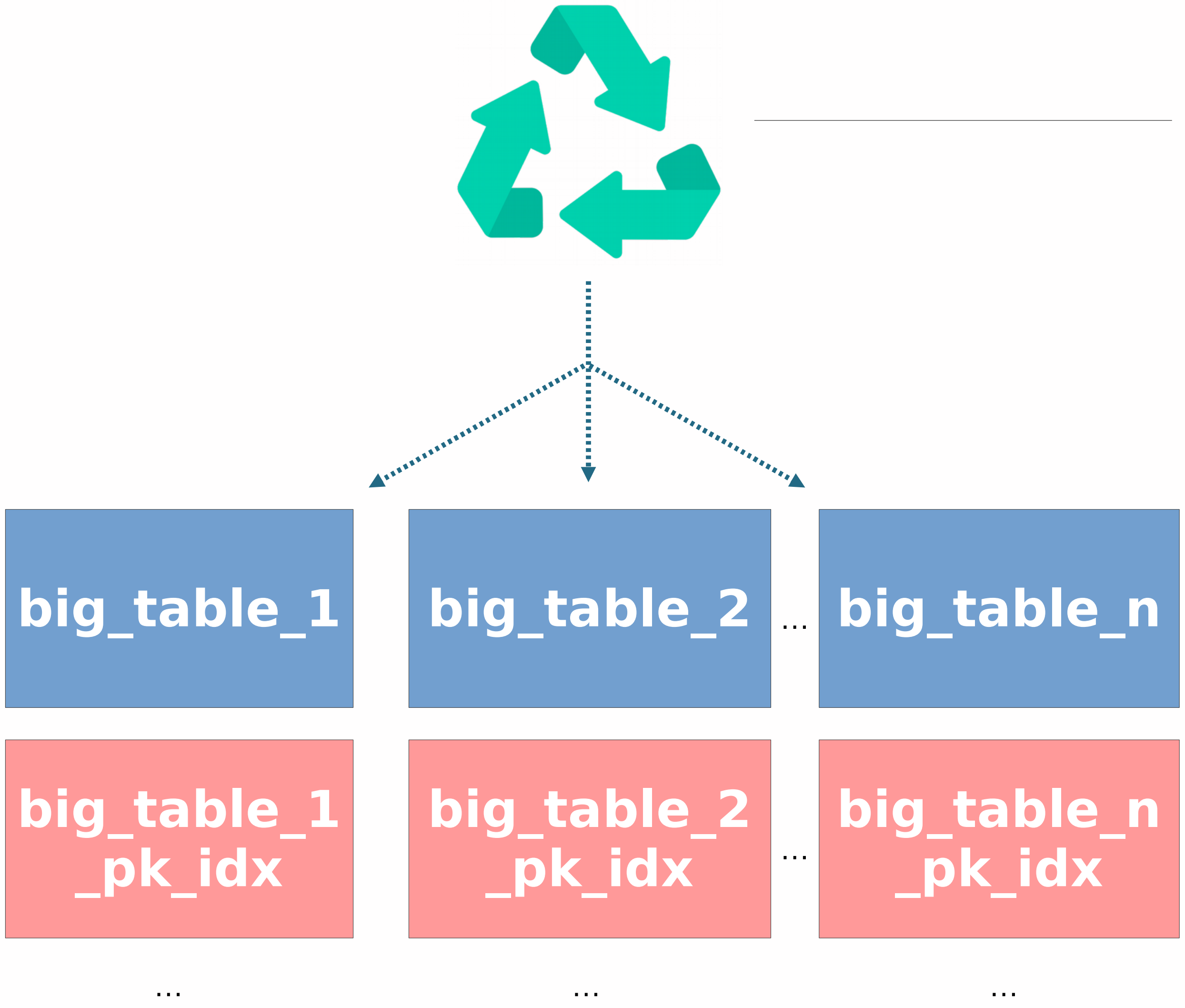
The naive approach



```sql
CREATE TABLE big_table_part_info
SELECT 'big_table_' || n AS name,
  n AS start, n + 200 AS next
FROM generate_series(0, 1800, 200)
  AS n
```

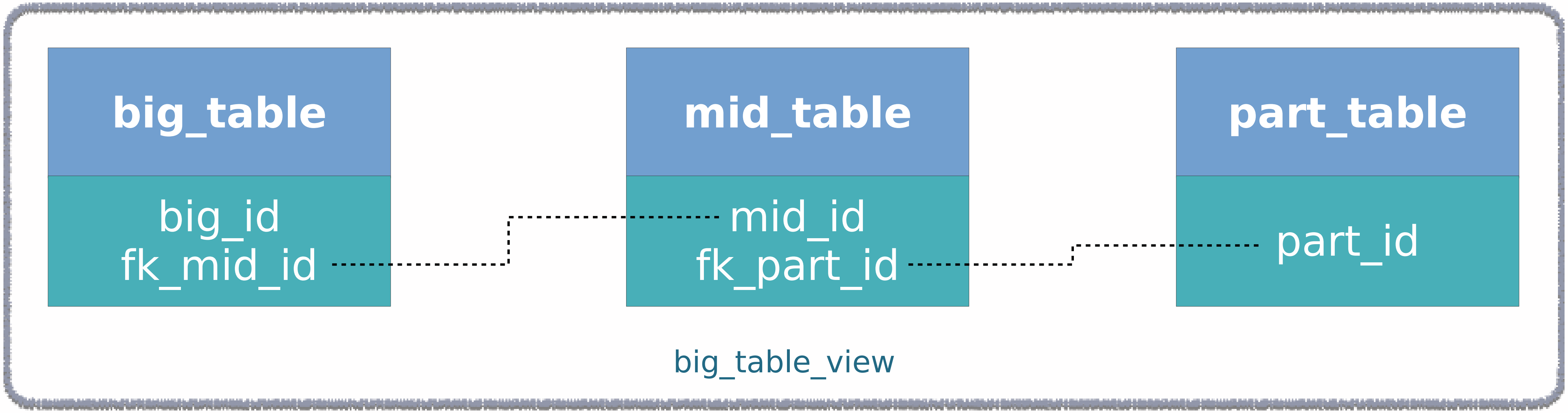
name	start	next
big_part_1	0	200
big_part_2	200	400
...	...	...
big_part_10	1800	2000

The naive approach



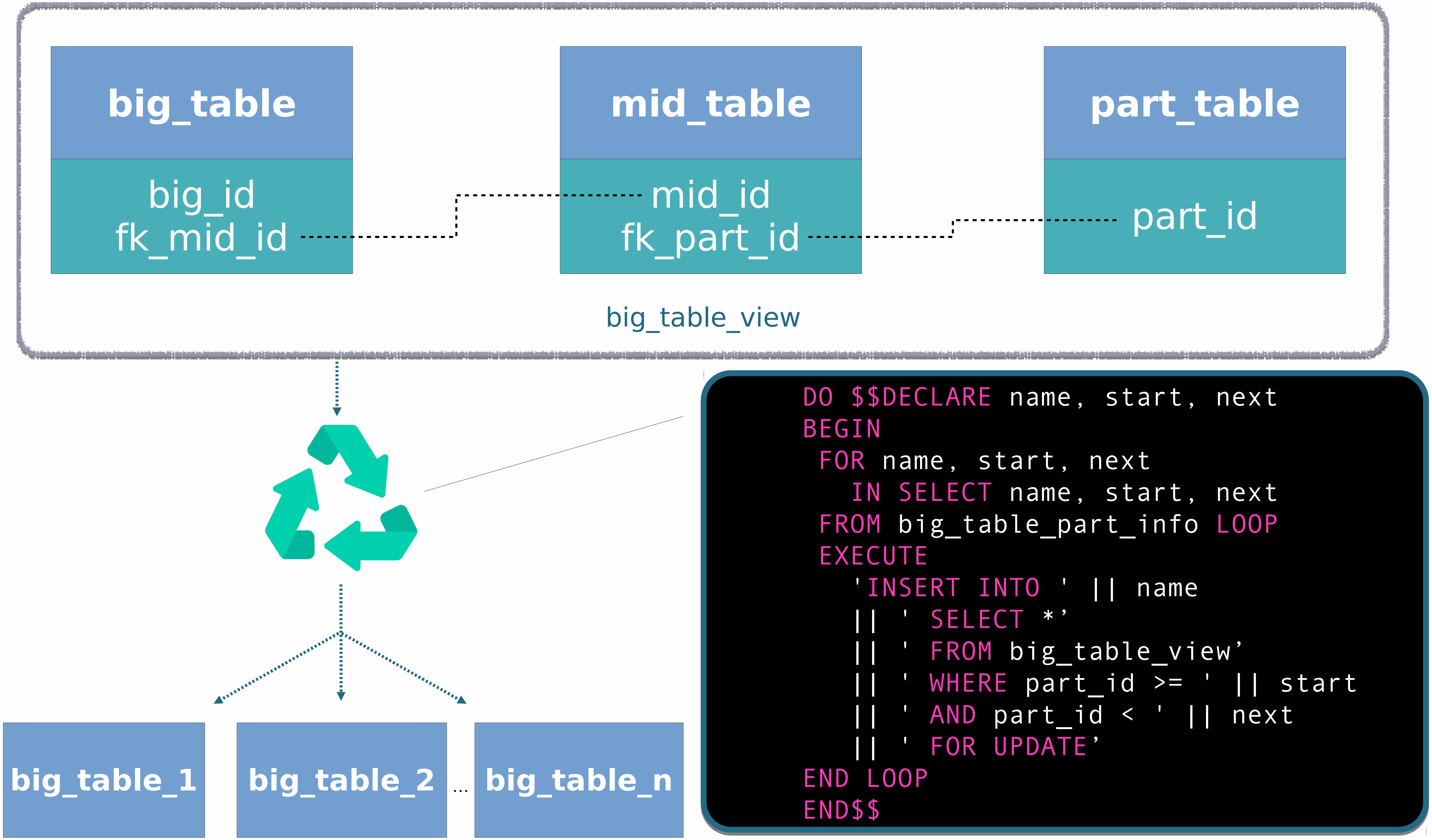
```
DO $$DECLARE name, start, next
BEGIN
 FOR name, start, next
   IN SELECT name, start, next
 FROM big_table_part_info LOOP
 EXECUTE
   'CREATE TABLE ' || name || '('
   || ' CHECK part_id >= ' || start
   || ' AND part_id < ' || next || ')'
   || ' INHERITS (big_table_part)’;
 EXECUTE
   'CREATE INDEX ' || name || '_pk_idx'
   || ' ON ' || name || '(big_id)’;
  …
END LOOP;
END$$
```

The naive approach



```
SELECT p.part_id, b.*
FROM big_table AS b
JOIN mid_table ON (…)
JOIN part_key_table AS p ON (...)
```

The naive approach



```
DO $$DECLARE name, start, next
BEGIN
 FOR name, start, next
   IN SELECT name, start, next
 FROM big_table_part_info LOOP
 EXECUTE
   'INSERT INTO ' || name
   || ' SELECT *'
   || ' FROM big_table_view’
   || ' WHERE part_id >= ' || start
   || ' AND part_id < ' || next
   || ' FOR UPDATE’
END LOOP
END$$
```

The naive approach

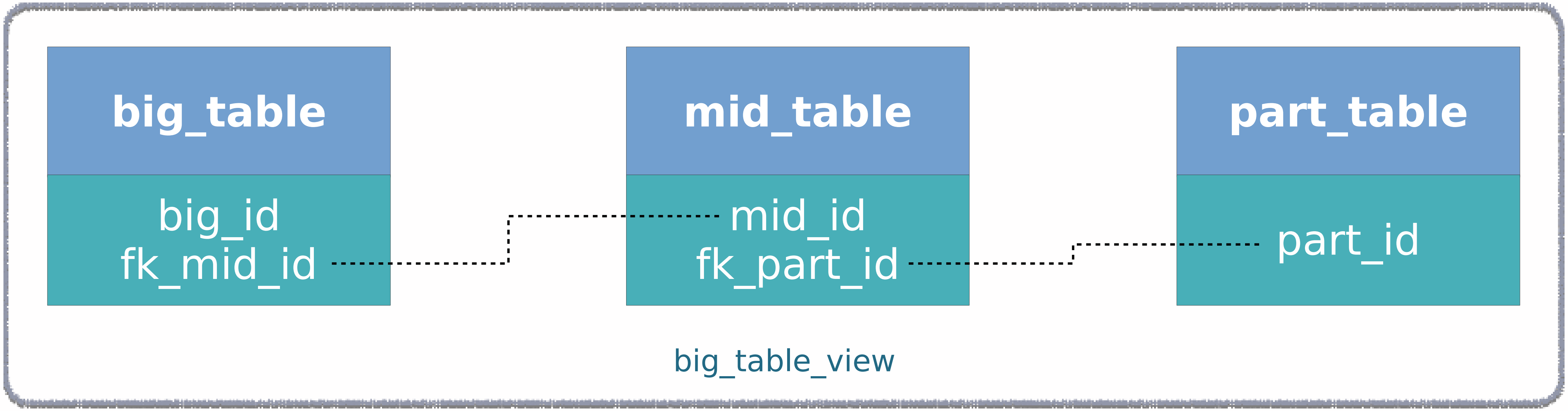
- ✓ 2 days maintenance window (will be enough!) 1
- ✓ big_table empty copy plus partition key 2
- ✓ Homogeneous ranges of partition key 3
- ✓ ~~Copy directly to partitions from ad-hoc view~~ 4

WRONG! It takes too long, full scan 3 tables repeated per partition (~8 hour each)

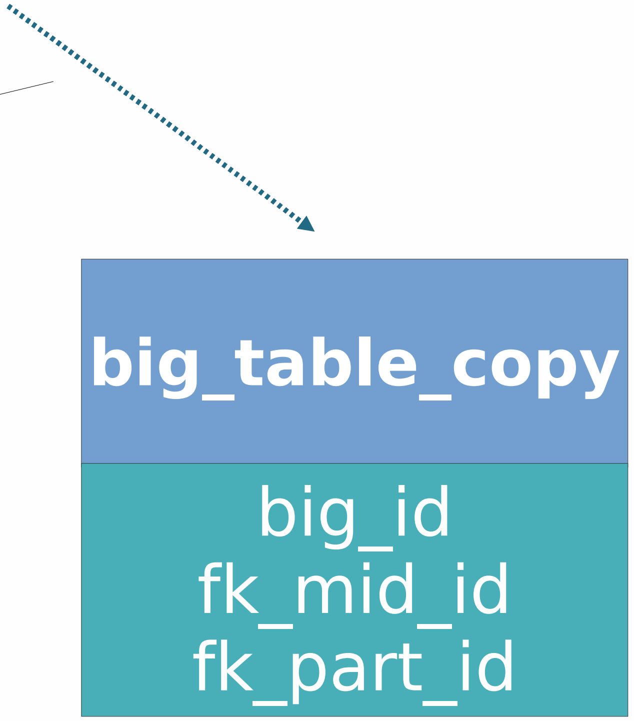
The tuned approach

- ✓ 2 days maintenance window (will be enough!) 1
- ✓ big_table empty copy plus partition key 2
- ✓ Copy of big_table from ad-hoc view 3
- ✓ Homogeneous ranges of partition key 4
- ✓ Copy directly to partitions from ad-hoc view 5

The tuned approach

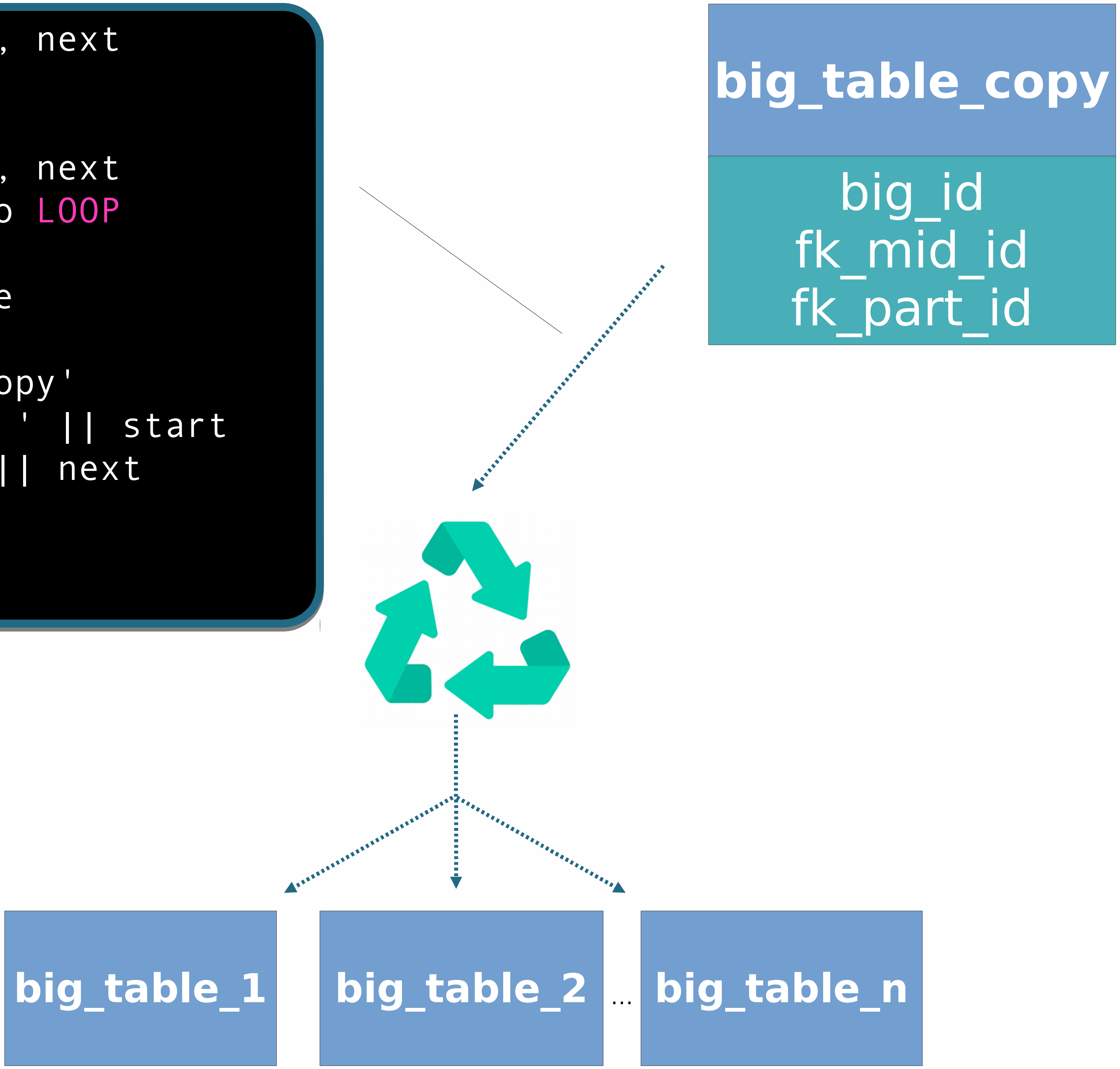


```
CREATE TABLE big_table_copy
  (fk_part_id integer)
INHERITS (big_table);
ALTER TABLE big_table_copy
NO INHERIT big_table;
INSERT INTO big_table_copy
SELECT * FROM big_table_view;
```



The tuned approach

```
DO $$DECLARE name, start, next
BEGIN
 FOR name, start, next
   IN SELECT name, start, next
 FROM big_table_part_info LOOP
 EXECUTE
   'INSERT INTO ' || name
   || ' SELECT *'
   || ' FROM big_table_copy'
   || ' WHERE part_id >= ' || start
   || ' AND part_id < ' || next
   || ' FOR UPDATE'
END LOOP
END$$
```



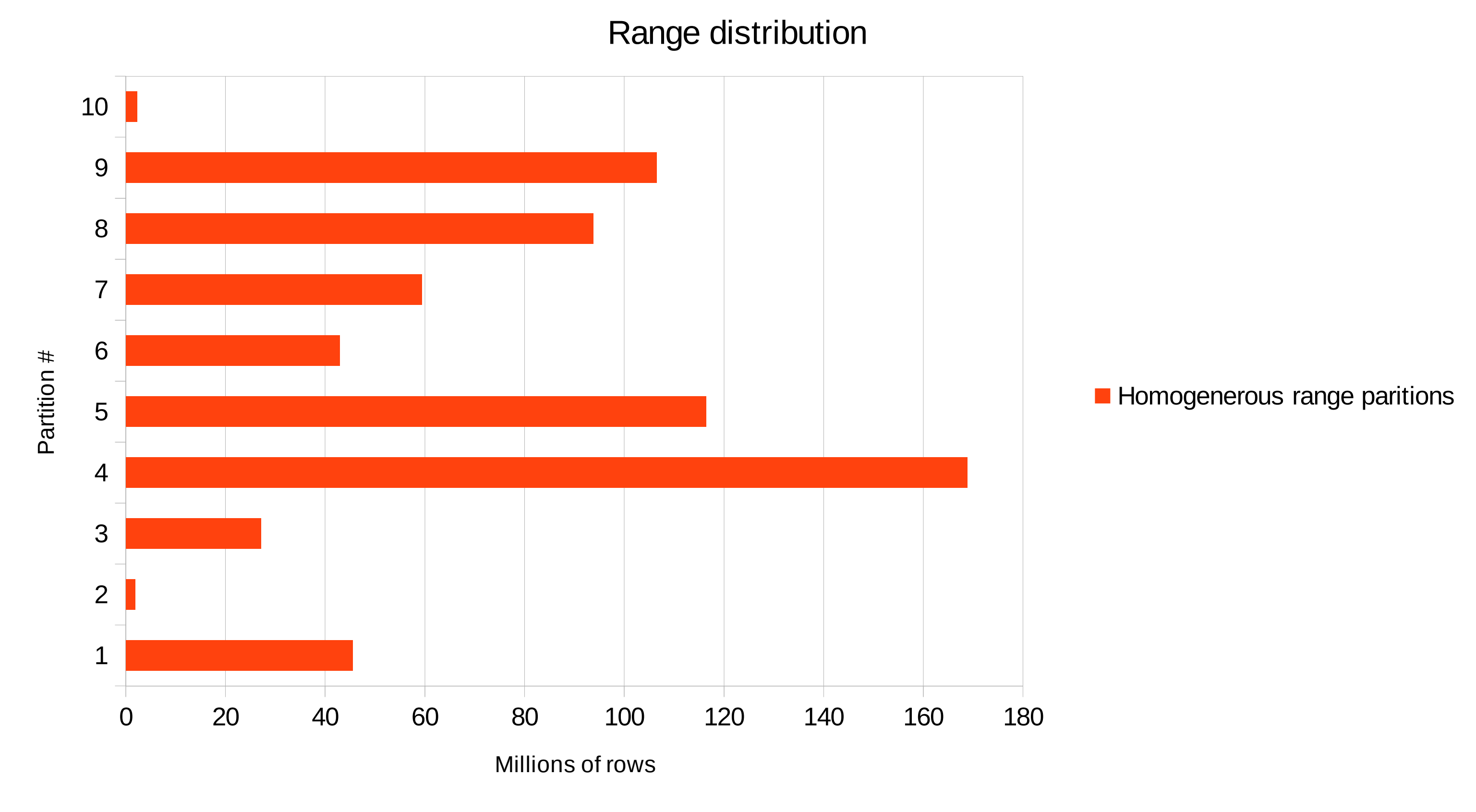
big_table_copy
big_id fk_mid_id fk_part_id

The tuned approach

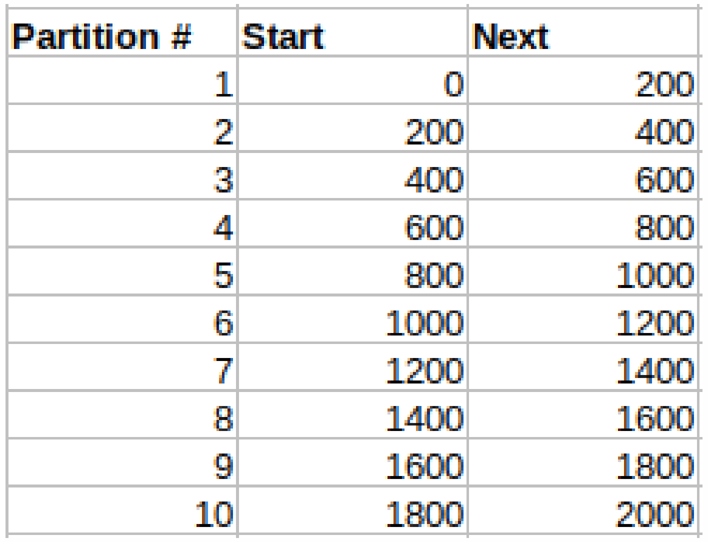
- ✓ 2 days maintenance window (will be enough!) 1
- ✓ big_table empty copy plus partition key 2
- ✓ Copy of big_table from ad-hoc view 3
- ✓ ~~Homogeneous ranges of partition key~~ 4
- ✓ Copy directly to partitions from ad-hoc view 5

WRONG! Homogeneous range <> Homogeneous data distribution

The tuned approach



Homogeneous range partitions



Partition #	Start	Next
1	0	200
2	200	400
3	400	600
4	600	800
5	800	1000
6	1000	1200
7	1200	1400
8	1400	1600
9	1600	1800
10	1800	2000

The smart approach

✓ Count group of rows 1

✓ 2 days maintenance window (will be enough!) 2

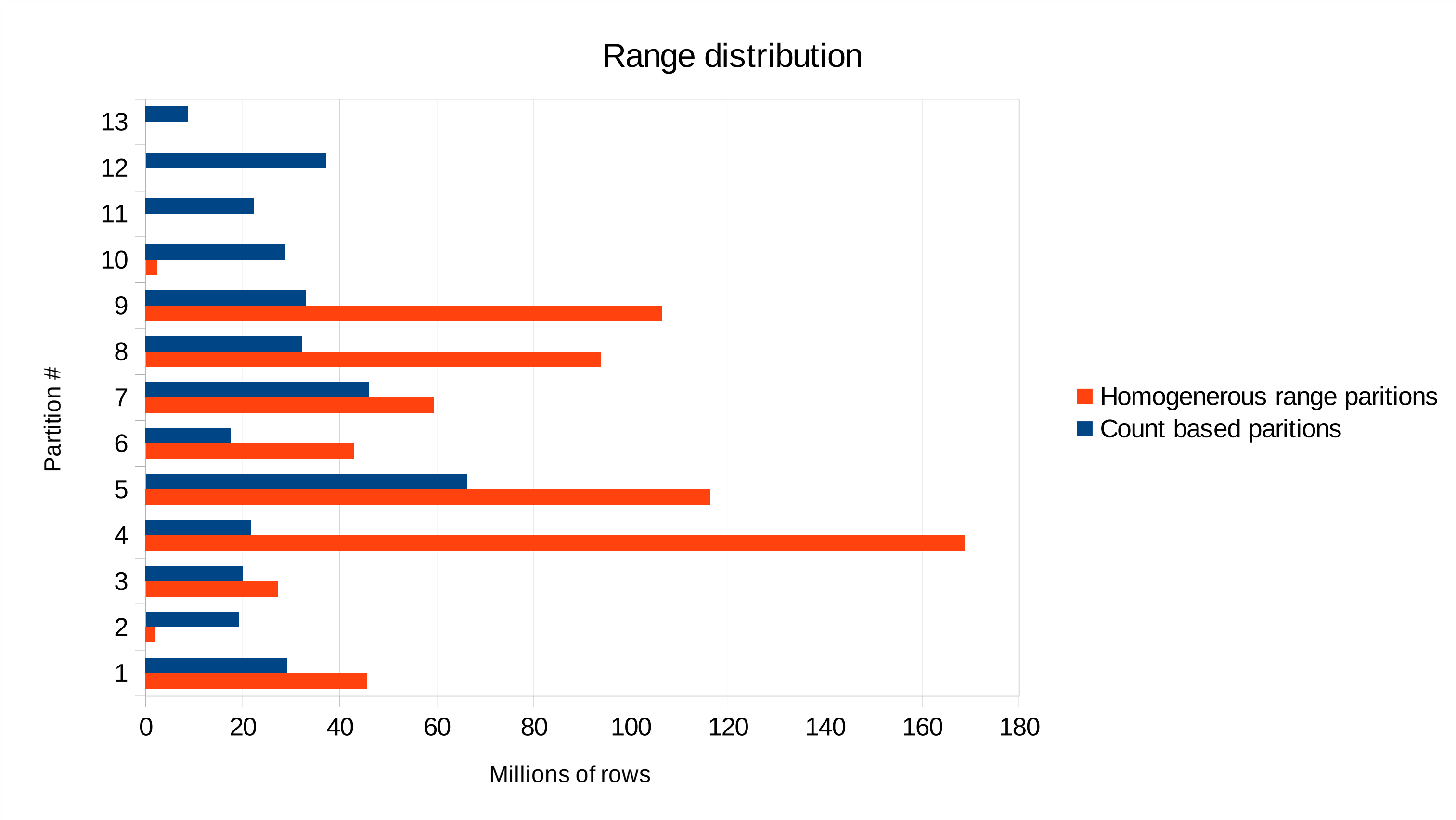
✓ big_table empty copy plus partition key 3

✓ Copy of big_table plus partition key 4

✓ Partition by ranges based on count of partition key 5

✓ Copy directly to partitions from ad-hoc view 6

The smart approach



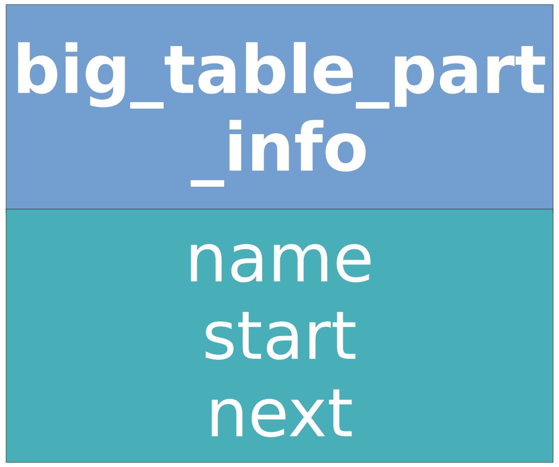
Homogeneous range partitions

Partition #	Start	Next
1	0	200
2	200	400
3	400	600
4	600	800
5	800	1000
6	1000	1200
7	1200	1400
8	1400	1600
9	1600	1800
10	1800	2000

Count based partitions

Partition #	Start	Next
1	0	100
2	100	200
3	200	500
4	500	705
5	705	710
6	710	800
7	800	820
8	820	900
9	900	1200
10	1200	1350
11	1350	1450
12	1450	1750
13	1750	1900

The smart approach

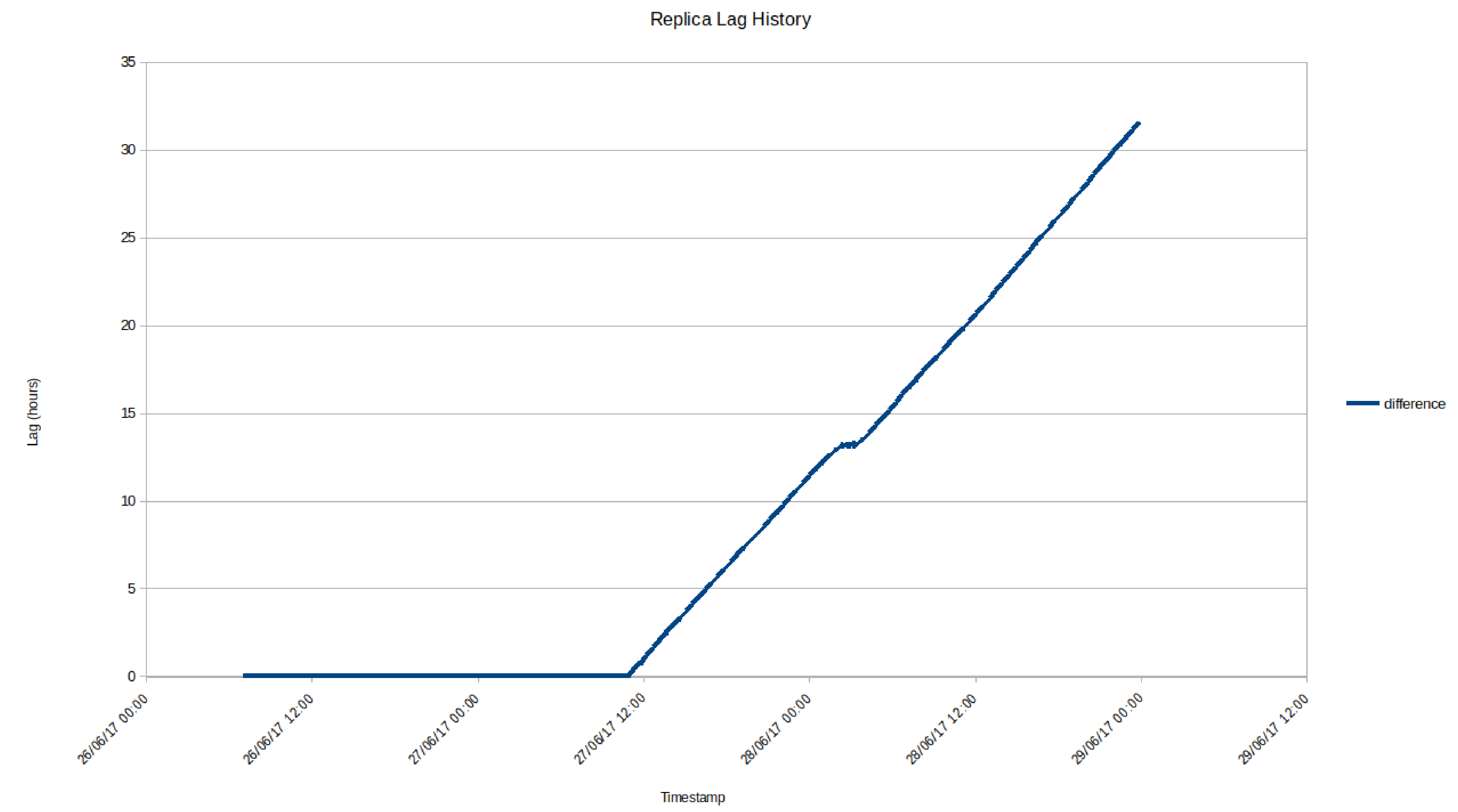


```
CREATE TABLE big_table_part_info AS
SELECT 'big_table_1' AS name,
  0 AS start, 100 AS next
UNION ALL
SELECT 'big_table_2' AS name,
  100 AS start, 200 AS next
UNION ALL
…
UNION ALL
SELECT 'big_table_10' AS name,
  1750 AS start, 1900 AS next
```

name	start	next
big_part_1	0	100
big_part_2	100	200
…	…	…
big_part_10	1750	1900

BINGO

The little problem



We forgot our replica!
Under heavy load, RDS replica seemed to stop replicating via network and switch to WAL shipping, which was extremely slow and lag grew to days!

Solving the little problem

Upgrade wal_keep_segments from 64 to 4096 so replica stay with SR

Nice idea but replica still get out of sync?!

Solving the little problem

Let’s create a new replica and remove the old one then!

But sometimes the replica could take a day to catch up.

Solving the little problem

How to make replica stay within SR?

Copy data by chunks
monitoring replication lag

Solving the little problem

How to make it in a window of 2 days?

Don’t do it in a window,
do it LIVE! lag

Do it LIVE!

- Application are not aware
  - Trick application to think it is using the real table! (INSTEAD OF to the rescue)
- Shit happens
  - Ability to pause/resume the process if needed
- Short enough resource usage duration
  - Be polite, do not starve resources
  - Also, don’t forget the little problem (lag monitoring)
- Preserve data consistency
  - Obviously

The live approach

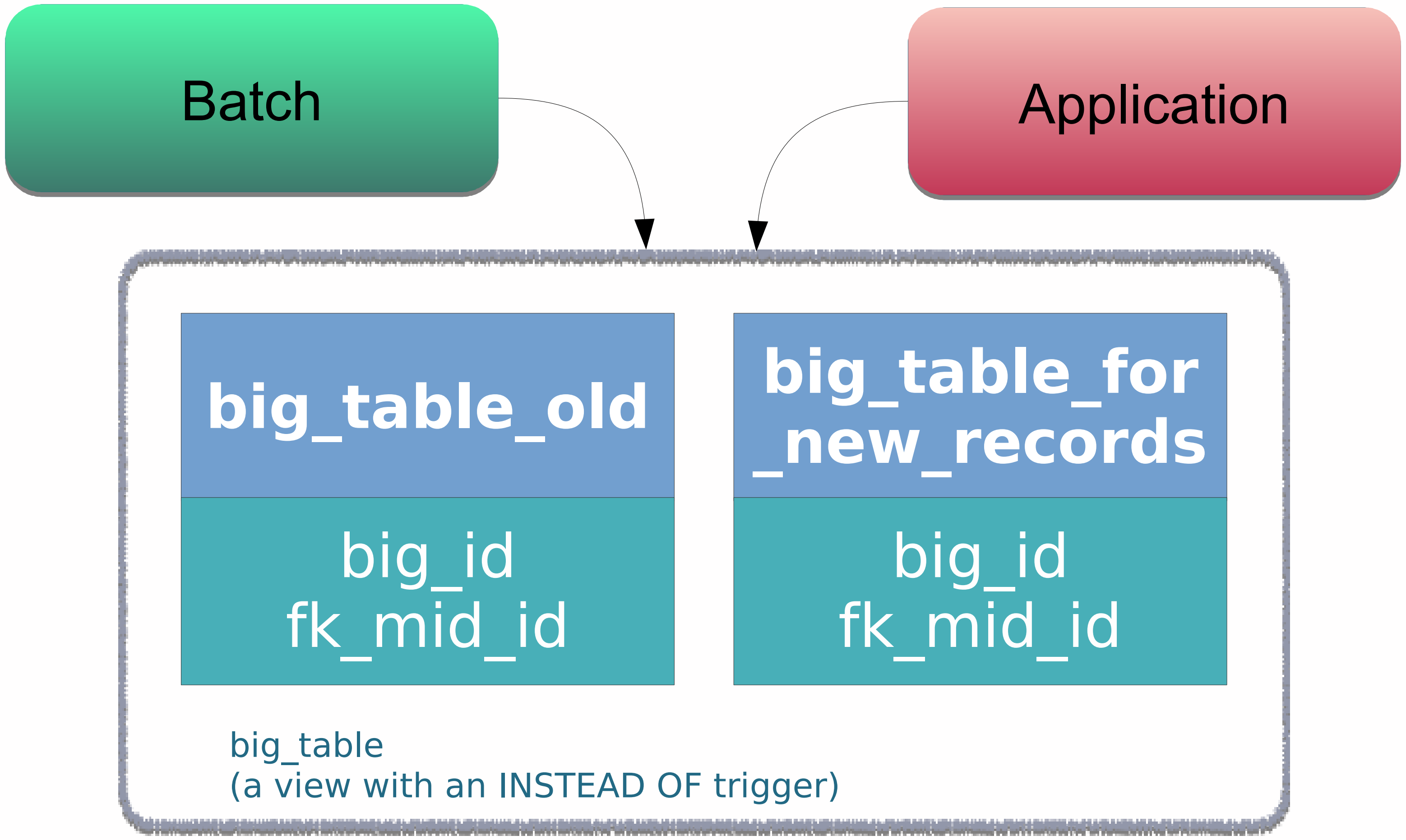
1. Count group rows
2. 8 hours maintenance window (will be enough!)
3. Indexes to help retrieve partition key faster
4. Empty copy of the table plus partition key
5. Partition by range based on count of partition key
6. Create helper table for the new inserted rows
7. Rename big_table and create a fake view with trigger to handle changes
8. Copy data to the partitions by chunks of 1M rows (LIVE part)

The live approach - the fake view

Applications “thinks” they are using the real big_table…

…INSTEAD OF that, they’re accessing a view that fakes the real big_table. The view create an abstraction that allow to read real data and (with the help of a trigger) to modify real data.

The live approach - the fake view



Batch

Application

big_table_old

big_id
fk_mid_id

big_table_for_new_records

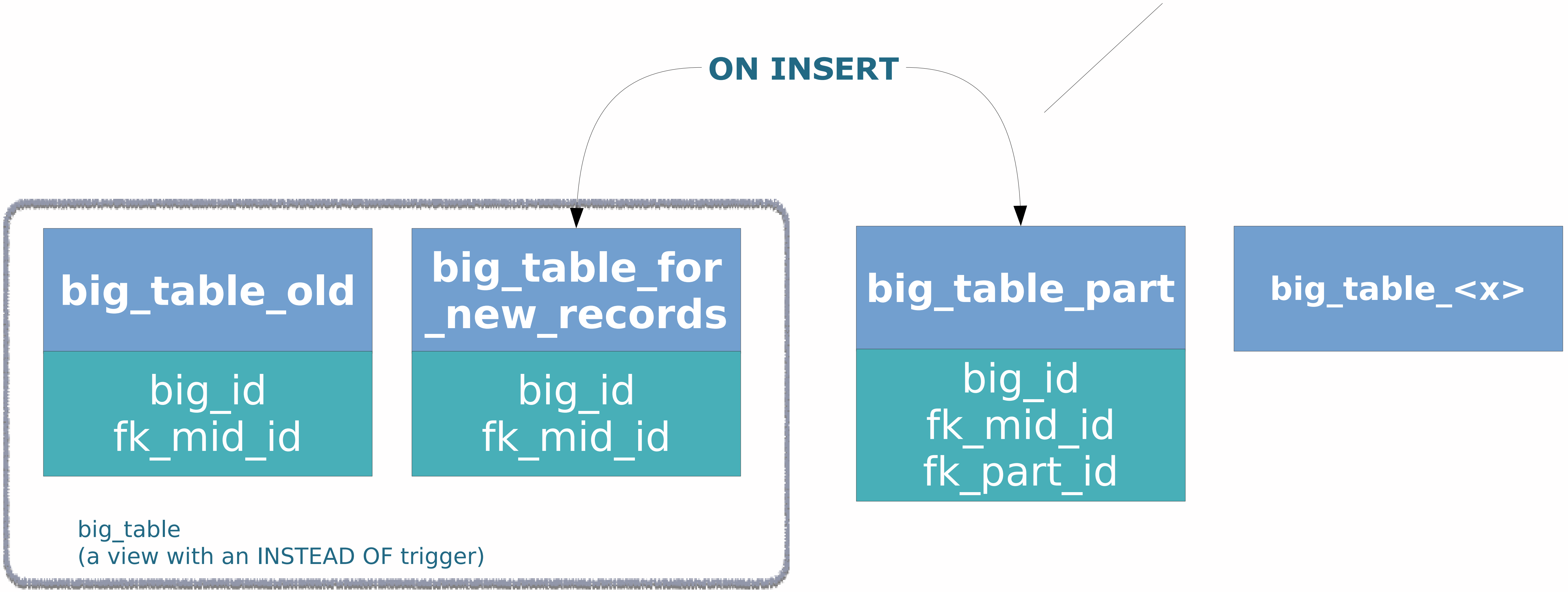
big_id
fk_mid_id

big_table
(a view with an INSTEAD OF trigger)

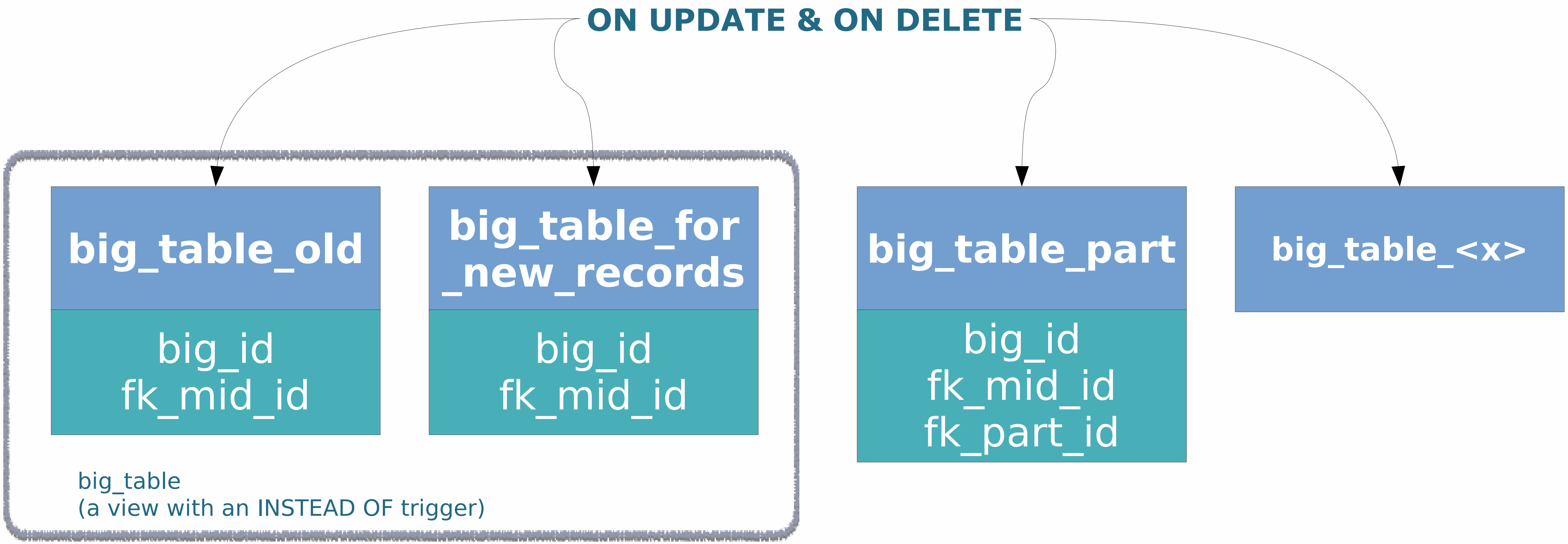
```
SELECT * FROM big_table_old
UNION ALL
SELECT * FROM big_table_fow_new_records
```

The live approach - the fake view

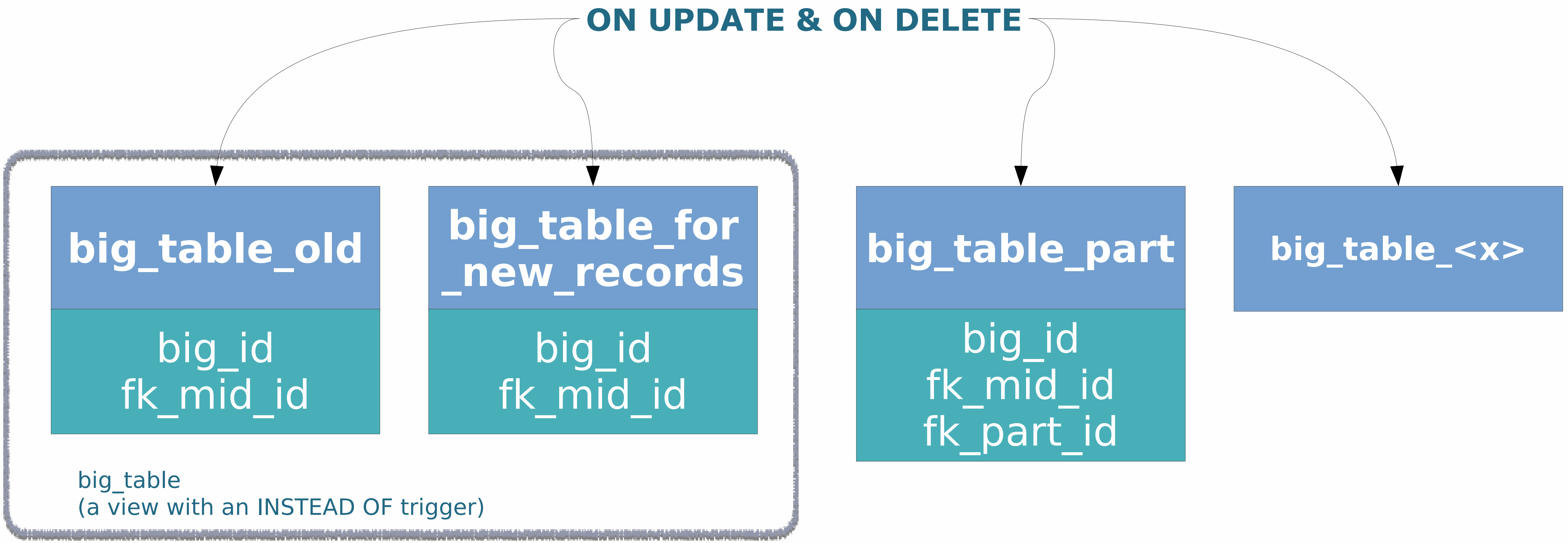
```
fk_part_id := (SELECT p.part_id
FROM mid_table AS m
JOIN part_key_table AS p ON (…)
WHERE mid_id = NEW.fk_mid_id)
```



The live approach - the fake view



The live approach - the fake view



ON UPDATE & ON DELETE

big_table_old
big_id
fk_mid_id

big_table_for_new_records
big_id
fk_mid_id

big_table
(a view with an INSTEAD OF trigger)

big_table_part
big_id
fk_mid_id
fk_part_id

big_table_<x>

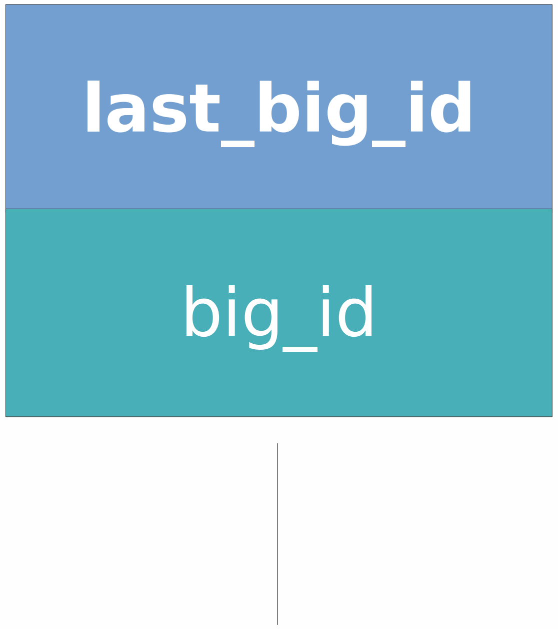
WARNING
ON UPDATE we forbid change
big_id and fk_part_id,
or the whole process
could break!!

The live approach - copy a chunk of data

Two step approach

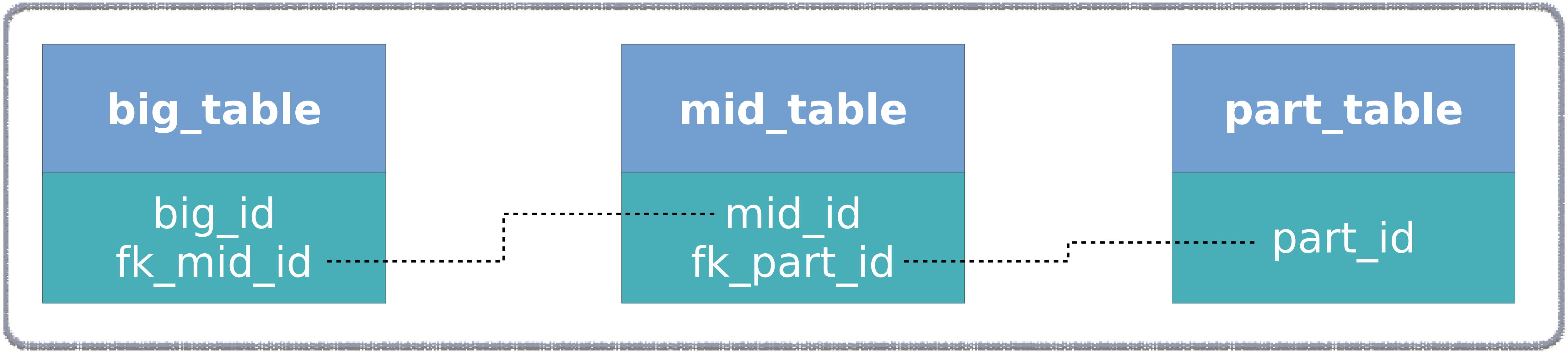
- ✓ Copy 1M rows to intermediate table — 1
- ✓ Move from intermediate table to partitions — 2

The live approach - copy a chunk of data

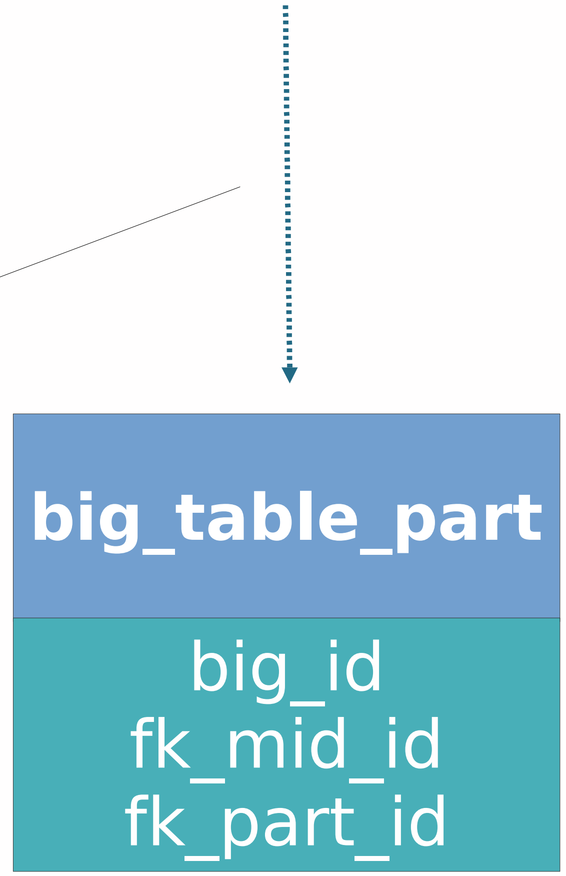


```sql
CREATE TABLE IF NOT EXISTS last_big_id(big_id bigint)
```

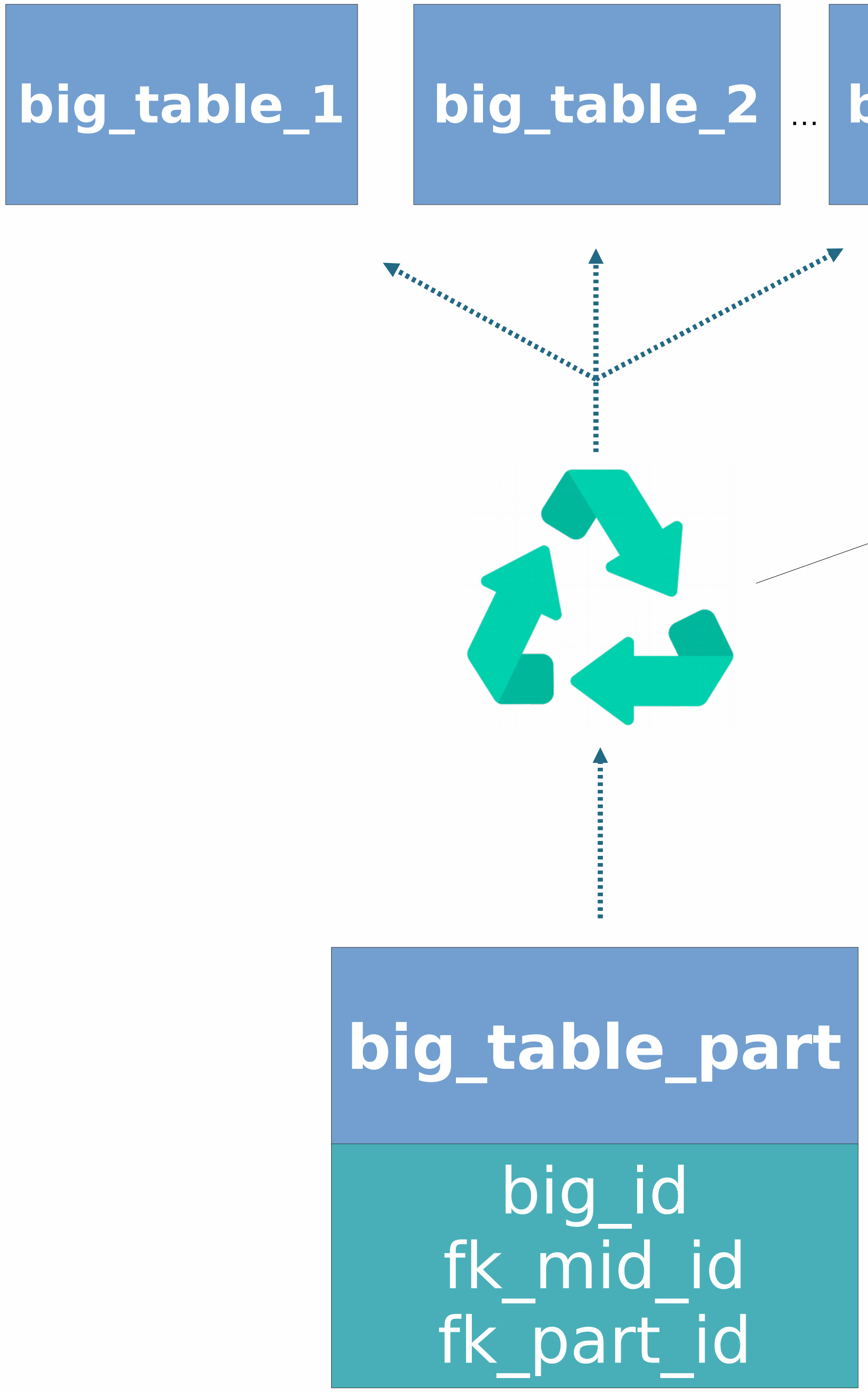
The live approach - copy a chunk of data



```
WITH big_table_chunk AS
  (INSERT INTO big_table_part
  SELECT b.*, p.part_id
  FROM big_table_old AS b
  JOIN mid_table ON (…)
  JOIN part_key_table AS p ON (…)
  WHERE big_id > (
    SELECT coalesce(max(big_id),-1)
    FROM last_big_id)
  ORDER BY big_id LIMIT 1000000
  FOR UPDATE RETURNING big_id)
INSERT INTO last_big_id SELECT max(big_id)
FROM big_table_chunk
```



The live approach - copy a chunk of data



big_table_1

big_table_2

...

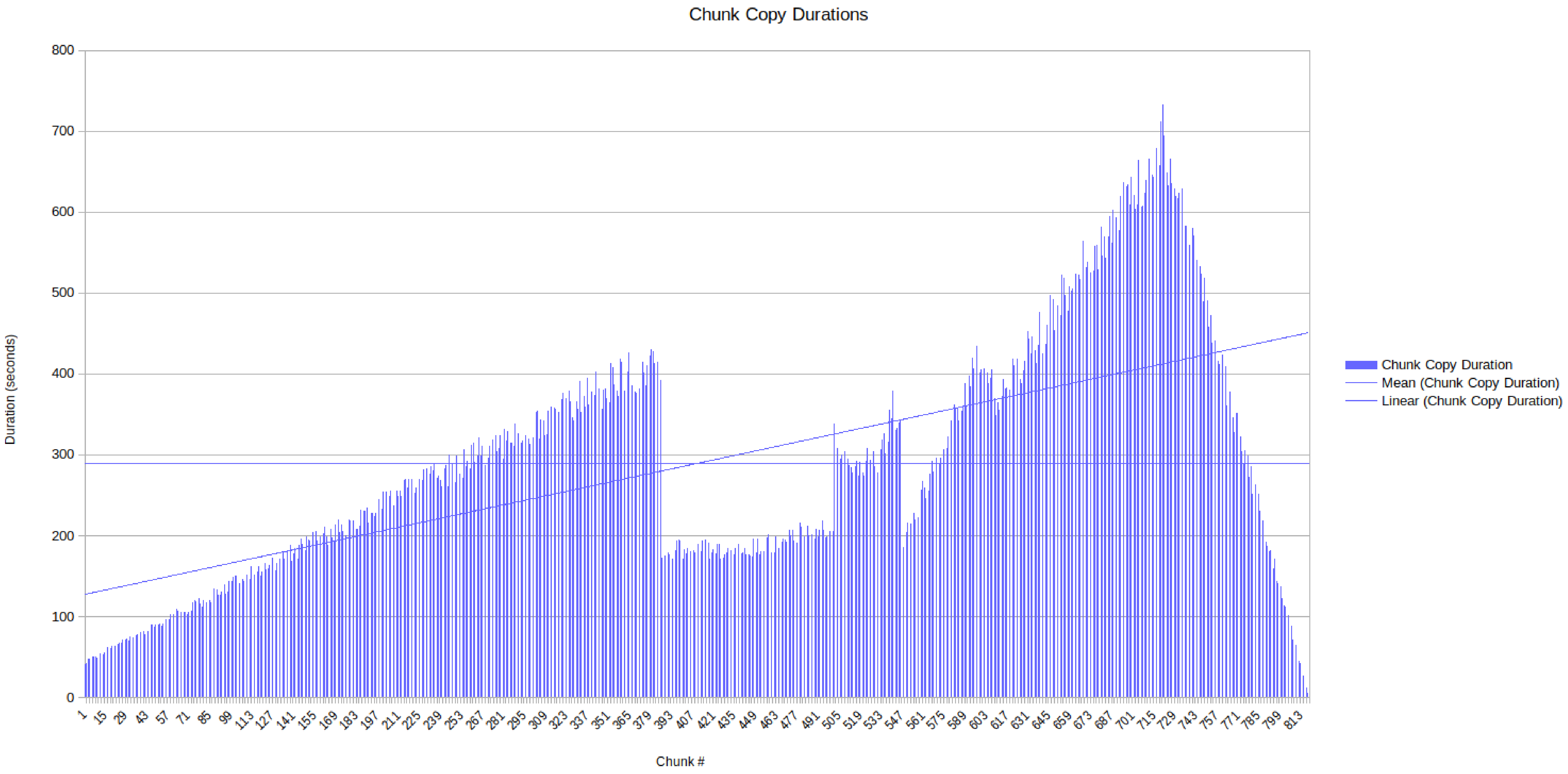
big_table_n

big_table_part

big_id
fk_mid_id
fk_part_id

```
DO $$DECLARE name, start, next
BEGIN
 FOR name, start, next
   IN SELECT name, start, next
 FROM big_table_part_info LOOP
   EXECUTE
   'INSERT INTO ' || name
   || ' SELECT *'
   || ' FROM big_table_part'
   || ' WHERE part_id >= ' || start
   || ' AND part_id < ' || next
   || ' FOR UPDATE’
END LOOP
TRUNCATE ONLY big_table_part;
END$$
```

The live approach - Some data!



Postgresql 10 partitioning

- A new way to create a partitions that simplify and remove risk of errors when doing it manually (beware you still have to create PKs, FKs and indexes manually)

```
CREATE TABLE big_table_1
PARTITION OF big_table_part
FOR VALUES FROM (1) TO (666);
```

- So, how to apply this live partitioning technique to postgresql 10?

Postgresql 10 partitioning

- Copy from view to parent table would fail since the parent table of a partition is not a real table:
  - We will need an intermediate table (big_table_inter) to hold the 1M rows readed from big_table_old
  - Also, when calculating last_big_id before copying a chunk we will have to do a:

```
SELECT max(big_id) FROM (
SELECT max(big_id) AS bid_ig FROM
big_table_part
  UNION ALL
SELECT max(big_id) AS big_id FROM
big_table_inter
) AS tmp
```

```
SELECT relname, relkind
  FROM pg_class
  WHERE relname
  LIKE 'big_table_%';
```

relname	relkind
big_table_part big_table_1	P r

Questions?