OpenSourceDB™
PostgreSQL MySQL Data Science

https://opensource-db.com/

Simplifying the Complexity of Postgres Partitioning

Hari P Kiran
Feb 22nd, 2023
PG Conf India

PGConf India
2023
WE'RE Back
22nd - 24th February 2023
Bengaluru

- Introduction
- Why partitioning?
- Inheritance
- Declarative Partitioning
- Partitioning in recent PG versions
- Pros/Cons of Partitioning
- Useful Tips & Tweaks
- Git public repo

What is not covered?

- Not everything and all-inclusive
- pg_partman
- Query access patterns
- Limitations of Not Partitioning

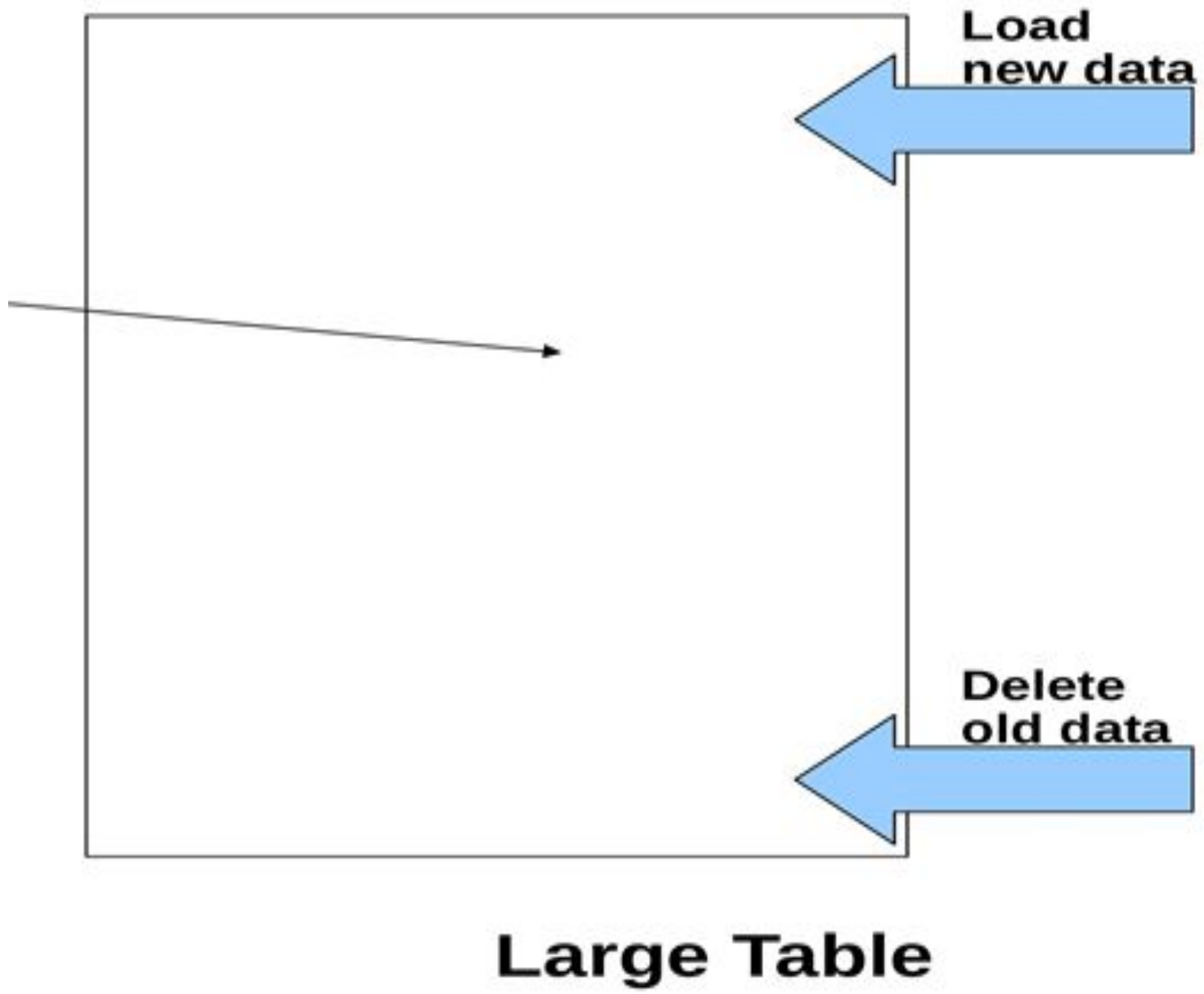
Introduction

- Partitioning refers to splitting what is logically one large table into smaller physical pieces.
- Present from PostgreSQL 8.1 version - table inheritance
- Certainly not a one-size-fits-all solution
- New implementation is called Declarative partitioning, starting from PostgreSQL 10
- Provides for faster queries of large tables
- Impact on I/O - reduction & distribution

Why Partitioning

Maintaining Historical tables

- COPY/INSERT new at front of table
- Most data read-only for long periods
- Deleting old data is expensive
    - DELETE FROM tbl WHERE date < X;
    - VACUUM



Partitioning: Data Maintenance

- Partition large tables
- SImplify data maintenance
- COPY/INSERT new data at front of table

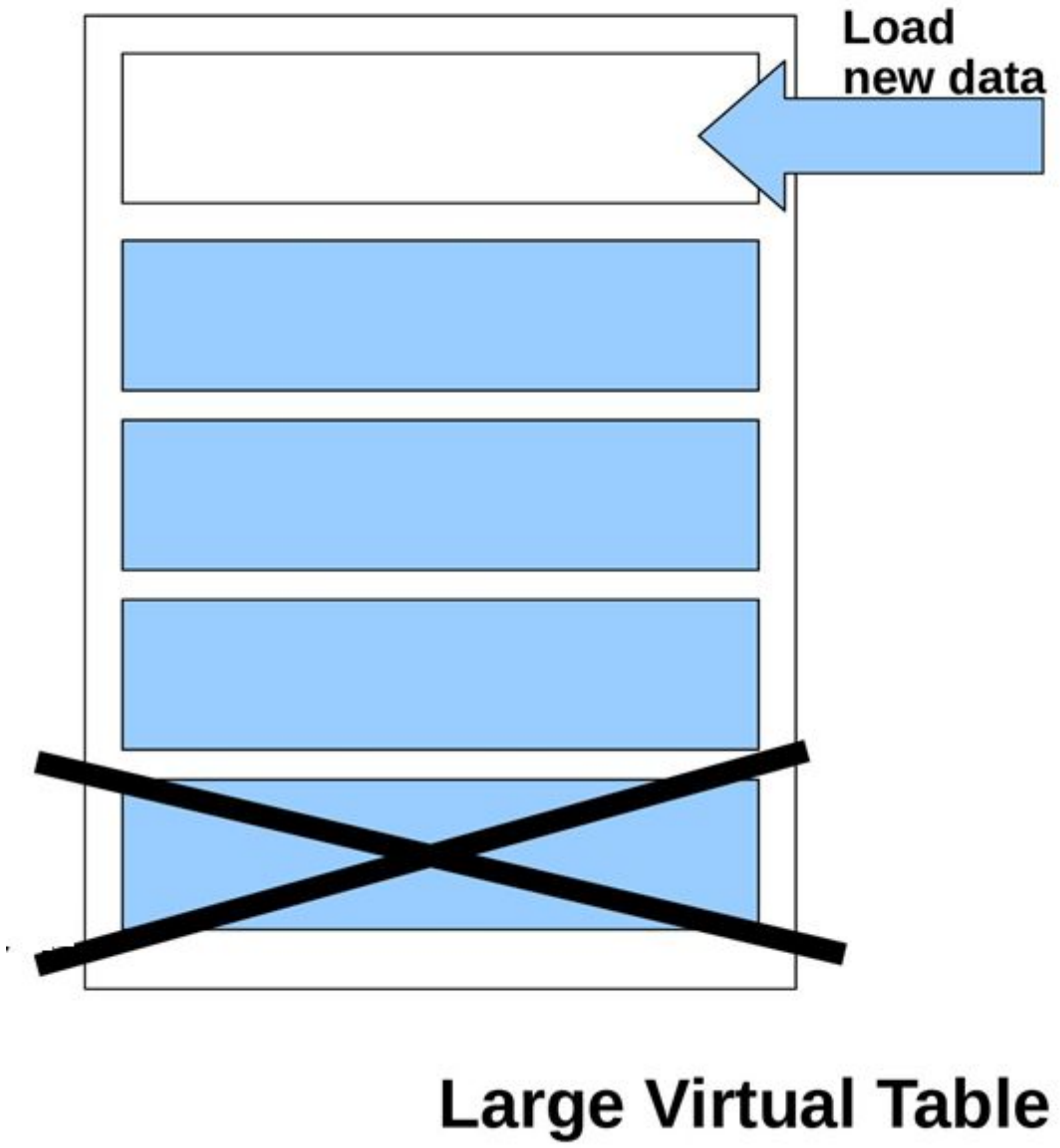
    -Use fast COPY

- Drop old tables

    -Better than DELETE

    -No need to VACUUM

- ADD/DROP partitions



Inheritance

-
  ```
  CREATE TABLE parent_table
   (Pkey integer not null,
    col2 char(1) not null,
    col3 text);
  ```

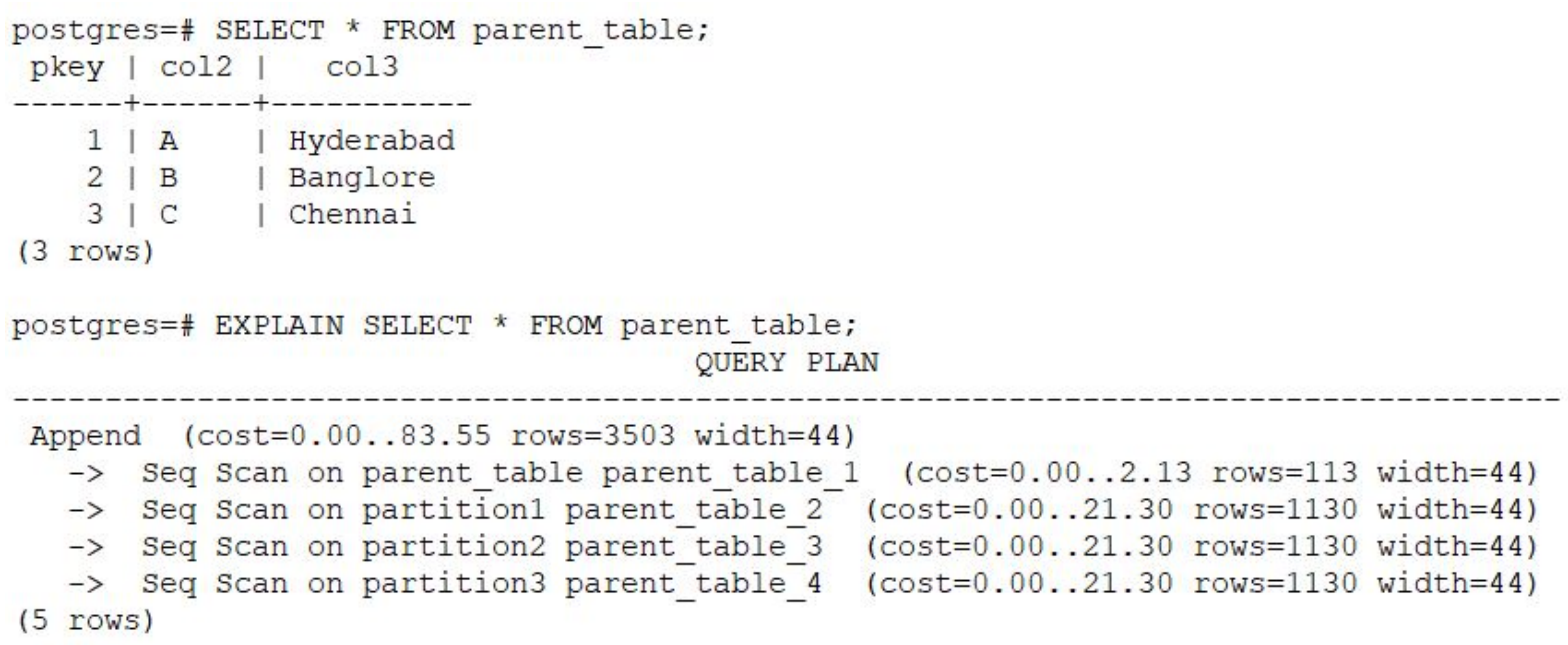
-
  ```
  CREATE TABLE partition1()
    INHERITS (parent_table);
  ```

-
  ```
  CREATE TABLE partition2()
    INHERITS (parent_table);
  ```

-
  ```
  CREATE TABLE partition3()
    INHERITS (parent_table);
  ```

- **Parent table is a template**
- **All “partitions” are just no-column tables.**
- **Partition columns are inherited from parent.**

Inheritance Select



```
postgres=# SELECT * FROM parent_table;
 pkey | col2 |    col3
------+------+-----------
    1 | A    | Hyderabad
    2 | B    | Banglore
    3 | C    | Chennai
(3 rows)

postgres=# EXPLAIN SELECT * FROM parent_table;
                                     QUERY PLAN
---------------------------------------------------------------------------------
 Append  (cost=0.00..83.55 rows=3503 width=44)
   ->  Seq Scan on parent_table parent_table_1  (cost=0.00..2.13 rows=113 width=44)
   ->  Seq Scan on partition1 parent_table_2  (cost=0.00..21.30 rows=1130 width=44)
   ->  Seq Scan on partition2 parent_table_3  (cost=0.00..21.30 rows=1130 width=44)
   ->  Seq Scan on partition3 parent_table_4  (cost=0.00..21.30 rows=1130 width=44)
(5 rows)
```

Inheritance Partitioning

- `CREATE TABLE parent_table (Pkey integer not null, Col2 char(1) not null, Col3 text);`
- `CREATE TABLE partition1 (CHECK (col2='A')) INHERITS (parent_table);`
- `CREATE TABLE partition2 (CHECK (col2='B')) INHERITS (parent_table);`
- `CREATE TABLE partition3 (CHECK (col2='C')) INHERITS (parent_table);`

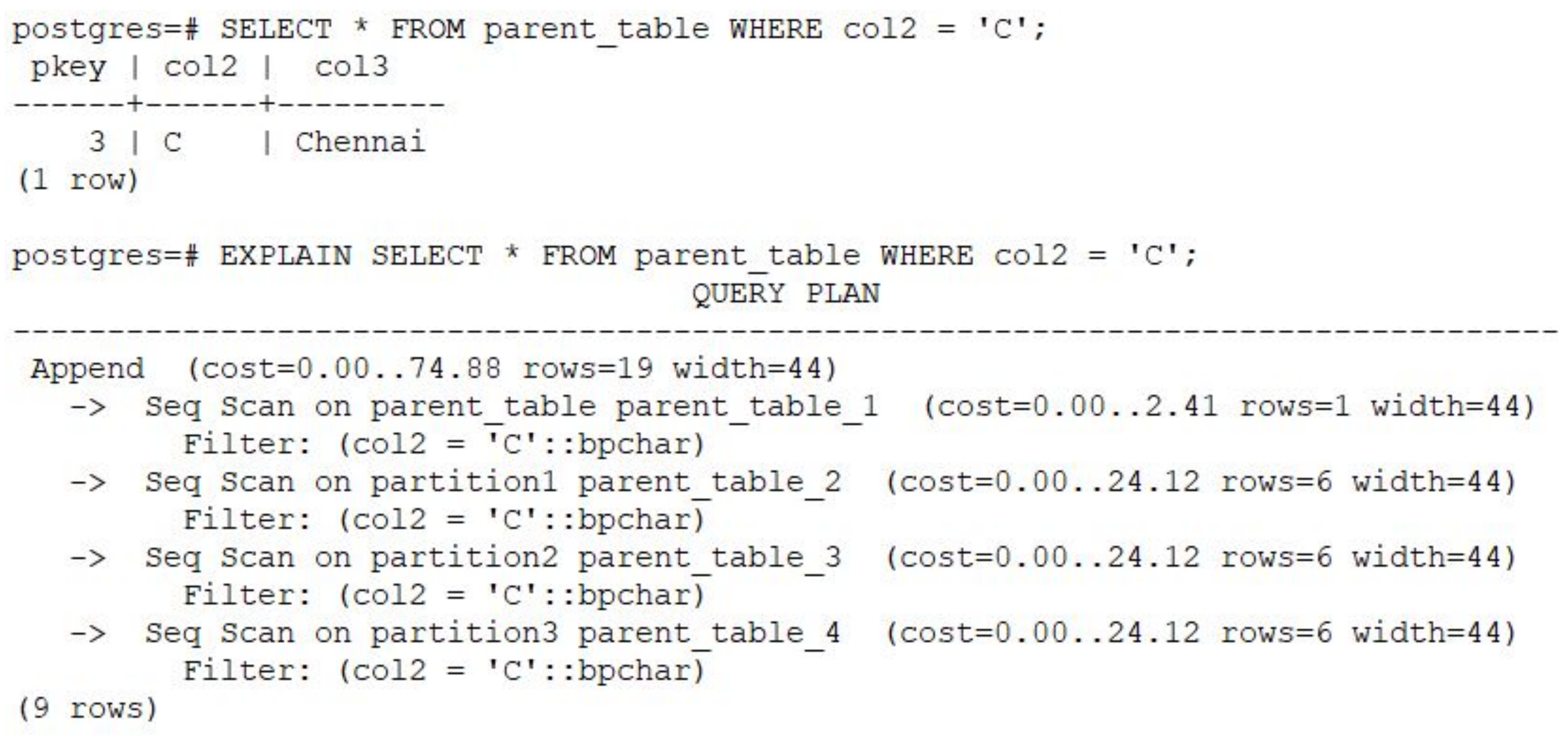
- **Non-overlapping CHECK constraints are defined on parent tables**
- **Don’t put any rows into parent table**
- **Don’t put any CHECK constraints on parent table**

Partition Planning

`constraint_exclusion='partition' (8.4+)`

- Simple plans allow better performance by removing partitions from plan that can be proven
- Flexible partitioning based on `CHECK` constraints
- `CHECK` constraints must be immutable
- Partitions should be distinct
- Can also be set to ‘on’ which allows `CHECK` constraints to be used for any table during planning, not just partitioning
  - Must be ‘on’ for useful partitioning before 8.4

Constraint Exclusion



```
postgres=# SELECT * FROM parent_table WHERE col2 = 'C';
 pkey | col2 |  col3
------+------+---------
    3 | C    | Chennai
(1 row)

postgres=# EXPLAIN SELECT * FROM parent_table WHERE col2 = 'C';
                                    QUERY PLAN
---------------------------------------------------------------------------------
 Append  (cost=0.00..74.88 rows=19 width=44)
   ->  Seq Scan on parent_table parent_table_1  (cost=0.00..2.41 rows=1 width=44)
         Filter: (col2 = 'C'::bpchar)
   ->  Seq Scan on partition1 parent_table_2  (cost=0.00..24.12 rows=6 width=44)
         Filter: (col2 = 'C'::bpchar)
   ->  Seq Scan on partition2 parent_table_3  (cost=0.00..24.12 rows=6 width=44)
         Filter: (col2 = 'C'::bpchar)
   ->  Seq Scan on partition3 parent_table_4  (cost=0.00..24.12 rows=6 width=44)
         Filter: (col2 = 'C'::bpchar)
(9 rows)
```

Constraint Exclusion Details

- `WHERE` clause must be *immutable*
  - `WHERE LogDate < '2023/01/01'` will work
  - `WHERE LogDate < now()` will not work

  because now() is a *stable* function

- Also means that we **cannot** use constraint exclusion for outside-in joins, e.g.

```
SELECT * FROM fact_tbl f
  JOIN date_dimension d ON (f.date_id = d.date_id)
  WHERE d.external_date < '2023/01/01'
```

Partition Good-to-knows

- Global Indexes are not supported
  - Would be very large and often unusable
  - Cause difficulty when dropping old partitions
- Partitioning works at planning time only
  - No partitions removed during joins
  - Must use constants inserted into query
  - Cannot prepare queries, uses one-time plans
  - No predicate push-down for LIMIT, ORDER BY etc.
  - MergeAppend plans available
    - Also used for MIN() and MAX()
  - Some join plans are worse than without partitioning

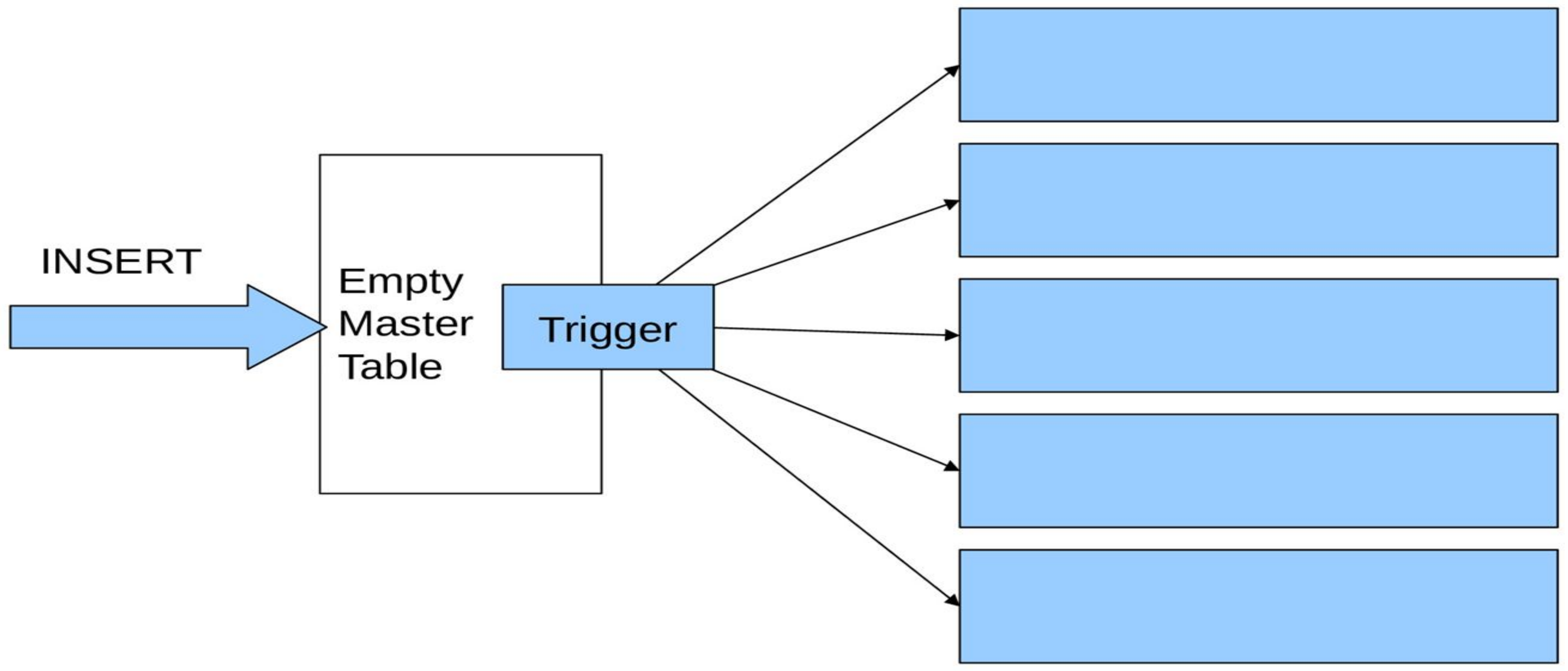
Benefits of Partitioning - Data Segregation

- Multiple check constraints allowed, so can work on multiple columns
- Check constraints can be any mix, so allows support for both range and list partitioning
- Indexes can be different on each table
- Storage options can differ for each table
  - Different tablespaces
  - Different compression
  - DIfferent stats and optimizer costs

Data Routing

- Automatic data routing for `INSERTs`
- Updates don't cause row movement between partitions
- `RULEs` cannot be used with `COPY`, so we cannot use a `RULE` to route data to partitions
- `TRIGGERs` can be used with `COPY`
- `AFTER` triggers use considerable memory
  - Performance problems on very large loads
  - Use carefully

Partitioning with Triggers - old school?



Child partition tables

Bulk data load

- `BEGIN;`
- `CREATE TABLE new_partition`

  `(LIKE ‘parent_table‘ INCLUDING DEFAULTS);`
  - Using LIKE means we don’t need to know exact table definition
- `COPY new_partition FROM 'data file';`
  - `COPY` in same transaction is very fast (only with `wal_level=replica`)
- `COMMIT;`

  `ALTER TABLE new_partition INHERIT parent_table;`

Declarative Partitioning

- New feature in PostgreSQL 10
- Internally still inheritance, but replaces
  - constraints => `LIST, RANGE`
  - routing triggers => automatic routing
- Flexible than inheritance partitioning
  - Maintain same table definition, different indexes
  - Triggers allow more complex schemes, but not needed
- Allows better query planning
  - The optimizer has insight into the schema

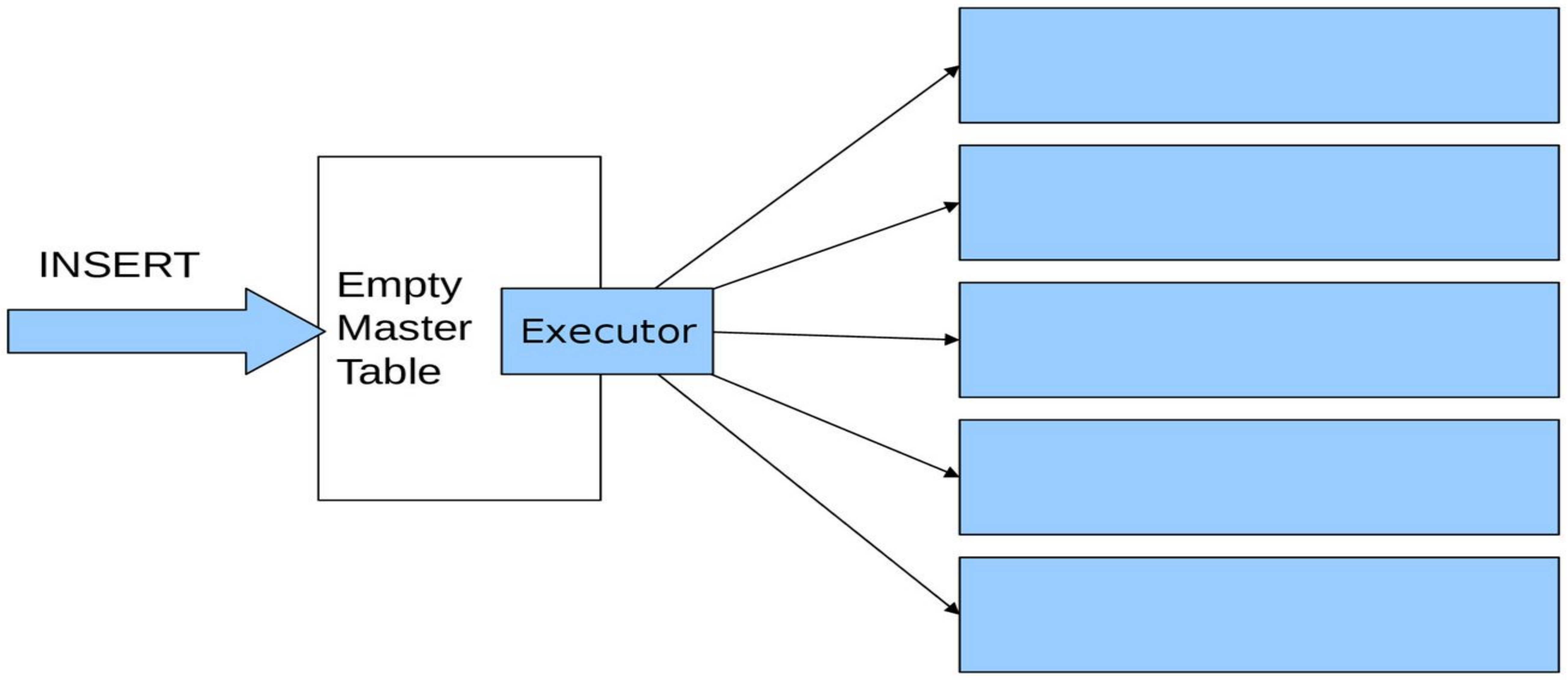
Declarative Partitioning - contd.

```
CREATE TABLE t (c date) PARTITION BY RANGE (c);

CREATE TABLE t_2023_01 PARTITION OF t FOR VALUES
   FROM (‘2023-01-01’) TO (’2023-02-01’);

CREATE TABLE t_2023_02 PARTITION OF t FOR VALUES
   FROM (’2023-02-01’) TO (’2023-03-01’);

CREATE TABLE t_2023_03 PARTITION OF t FOR VALUES
   FROM (’2023-03-01’) TO (’2023-04-01’);
```

Declarative Partitioning - cont.

- A few limitations similar to inheritance partitioning
- No global indexes
  - No primary or unique keys, or exclusion constraints
- No automatic movement between partitions
- Row triggers can be created on partitions

Partitioning using Metadata



Child partition tables

Partition Maintenance

- `CREATE TABLE t PARTITION BY ...;`
- `CREATE TABLE p AS PARTITION OF t FOR VALUES ...;`
- `ALTER TABLE t ATTACH PARTITION p FOR VALUES ...;`
    - Recommended to create a `CHECK` constraint matching the `FOR VALUES` clause first (eliminates validation scan while holding AccessExclusive lock on “t”)
- `DROP PARTITION p;`
- `ALTER TABLE t DETACH PARTITION p;`

Improvements and Features

- Declarative Partitioning Improvements
  - v11: Faster partition elimination
  - v11: Partition Pruning at Execution Time
- Dynamic Partition Elimination
- Implicit Partitioning
- BRIN: Automatic partitioning

Improvements and Features - contd.

- Declarative Partitioning Improvements
- Postgres 10: Brings in the original CREATE TABLE ... PARTITION BY ... declarative partitioning commands
- Postgres 11: Support for PRIMARY KEY, FOREIGN KEY, indexes, and triggers on partitioned tables.
- Postgres 11: INSERT on the parent partitioned table routes rows to their appropriate partition.
- Postgres 11: UPDATE statements can move rows between partitions.
- Postgres 12: Foreign keys can reference partitioned tables

Improvements and Features - contd.

- Postgres 12: Improved INSERT performance, ALTER TABLE ATTACH PARTITION no longer blocks queries.
- Postgres 13: Support for row-level BEFORE triggers on partitioned tables.
- Postgres 13: Logical replication on partitioned tables (previously, partitions would have to be replicated individually).
- Postgres 14: Partitions can be detached in a non-blocking way with ALTER TABLE ... DETACH PARTITION ... CONCURRENTLY
- Postgres 15: Improved planning time for statements where only few of the partitions are relevant
- Postgres 15: CLUSTER on partition tables

Faster Partition Elimination

author	Alvaro Herrera <alvherre@alvh.no-ip.org>
	Fri, 6 Apr 2018 19:23:04 +0000 (16:23 -0300)
committer	Alvaro Herrera <alvherre@alvh.no-ip.org>
	Fri, 6 Apr 2018 19:44:05 +0000 (16:44 -0300)
commit	9fdb675fc5d2de825414e05939727de8b120ae81
tree	0e599089ca1e82dac50b61675f1cdf2d53bb0b49 tree
parent	11523e860f8fe29f9142fb63c44e01cd0d5e7375 commit | diff

Faster partition pruning

Add a new module backend/partitioning/partprune.c, implementing a more sophisticated algorithm for partition pruning.

The new module uses each partition's "boundinfo" for pruning instead of constraint exclusion,based on an idea proposed by Robert Haas of a "pruning program": a list of steps generated from the query quals which are run iteratively to obtain a list of partitions that must be scanned in order to satisfy those quals.

At present, this targets planner-time partition pruning, but there exist further patches to apply partition pruning at execution time as well. This commit also moves some definitions from include/catalog/partition.h to a new file include/partitioning/partbounds.h, in an attempt to rationalize partitioning related code.
https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=9fdb675fc5d2de825414e05939727de8b120ae81

Partition Pruning at Execution Time

author Alvaro Herrera <alvherre@alvh.no-ip.org>

Sat, 7 Apr 2018 20:54:31 +0000 (17:54 -0300)

committer Alvaro Herrera <alvherre@alvh.no-ip.org>

Sat, 7 Apr 2018 20:54:39 +0000 (17:54 -0300)

commit 499be013de65242235ebdde06adb08db887f0ea5

tree c1f69b818f917379fb4f72e80535fef899a40b5b tree

parent 5c0675215e153ba1297fd494b34af2fdebd645d1 commit | diff

Support partition pruning at execution time

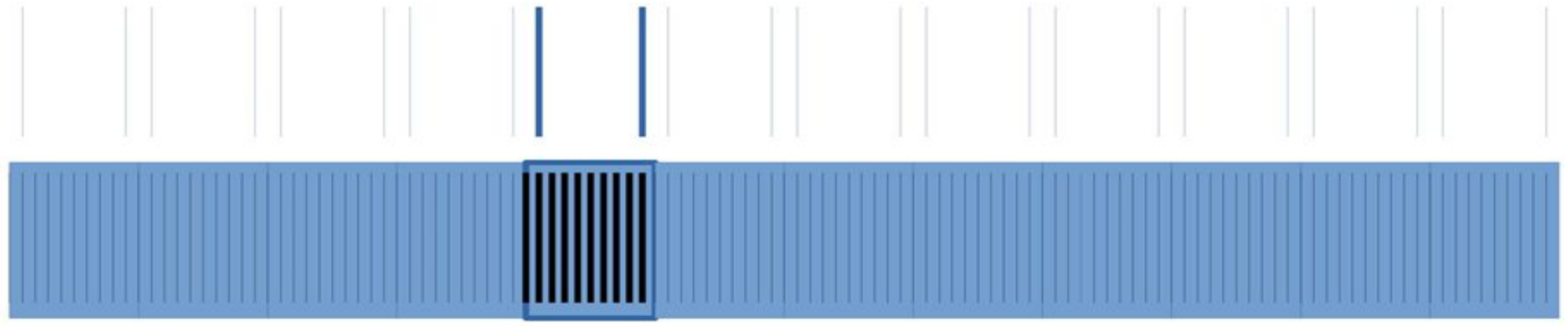
Existing partition pruning is only able to work at plan time, for query quals that appear in the parsed query. This is good but limiting, as there can be parameters that appear later that can be usefully used to further prune partitions.This commit adds support for pruning subnodes of Append which cannot possibly contain any matching tuples, during execution, by evaluating Params to determine the minimum set of subnodes that can possibly match. We support more than just simple Params in WHERE clauses. Support additionally includes:

1. Parameterized Nested Loop Joins: The parameter from the outer side of the join can be used to determine the minimum set of inner side partitions to scan.

2. Initplans: Once an initplan has been executed we can then determine which partitions match the value from the initplan.
https://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=499be013de65242235ebdde06adb08db887f0ea5

Block Range Indexes (BRIN)

- “Automatic Partitioning” or Poor man partitioning
  - Store min and max tuples for each page range
  - Use theorem proving to skip sections of seqscan
  - Can be added easily to existing applications



Block Range Indexes - contd.

Example:

```
CREATE TABLE test_brin
(id int NOT NULL PRIMARY KEY,
date TIMESTAMP NOT NULL,
level INTEGER,
msg TEXT);

INSERT INTO test_brin (id, date, level, msg)
   SELECT g,
   CURRENT_TIMESTAMP + ( g || 'minute' ) :: interval,
   random() * 6,
   md5(g::text)
   FROM generate_series(1,8000000) as g;
```

Block Range Indexes - contd.

```
EXPLAIN ANALYZE SELECT * from public.test_brin where date between '2023-02-19 14:30:44'
and '2023-02-19 14:55:44';
                                              QUERY PLAN
------------------------------------------------------------------------------------------

 Gather  (cost=1000.00..133475.57 rows=1 width=49) (actual time=8.196..1378.458 rows=25
loops=1)

   Workers Planned: 2

   Workers Launched: 1

   ->  Parallel Seq Scan on test_brin  (cost=0.00..132475.47 rows=1 width=49) (actual
time=451.658..1135.423 rows=12 loops=2)

         Filter: ((date >= '2023-02-19 14:30:44'::timestamp without time zone) AND (date
<= '2023-02-19 14:55:44'::timestamp without time zone))

         Rows Removed by Filter: 3999988

 Planning Time: 0.128 ms

 Execution Time: 1379.013 ms
```

Block Range Indexes - contd.

```
CREATE INDEX test_brin__date_idx on test_brin(date);     #by default B-Tree index


partition=# EXPLAIN ANALYZE SELECT * from public.test_brin where date between
'2023-02-19 14:30:44' and '2023-02-19 14:55:44';
                                  QUERY PLAN
------------------------------------------------------------------------------------

 Index Scan using "test_brin__date_idx " on test_brin  (cost=0.43..8.99 rows=28
width=49) (actual time=0.020..0.026 rows=25 loops=1)

   Index Cond: ((date >= '2023-02-19 14:30:44'::timestamp without time zone) AND

     (date <= '2023-02-19 14:55:44'::timestamp without time zone))

 Planning Time: 0.386 ms

 Execution Time: 0.053 ms
```

Block Range Indexes - contd.

```
CREATE INDEX test_brin_date_brin_idx on test_brin using brin (date);


\di+ test_brin_date_brin_idx
                                        List of relations
 Schema |          Name           | Type  |  Owner   |   Table   | Persistence | Access method | Size  |
Description
--------+-------------------------+-------+----------+-----------+-------------+---------------+-------+---------
 public | test_brin_date_brin_idx | index | postgres | test_brin | permanent   | brin          | 64 kB |
(1 row)
```

Block Range Indexes - contd.

```
EXPLAIN ANALYZE SELECT * from public.test_brin where date between '2023-02-19 14:30:44' and '2023-02-19
14:55:44';

                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------

 Bitmap Heap Scan on test_brin  (cost=20.03..33406.84 rows=1 width=49) (actual time=0.272..3.458 rows=25
loops=1)

   Recheck Cond: ((date >= '2023-02-19 14:30:44'::timestamp without time zone) AND (date <= '2023-02-19
14:55:44'::timestamp without time zone))

   Rows Removed by Index Recheck: 12391

   Heap Blocks: lossy=128

   ->  Bitmap Index Scan on test_brin_date_brin_idx  (cost=0.00..20.03 rows=12403 width=0) (actual
time=0.240..0.240 rows=1280 loops=1)

         Index Cond: ((date >= '2023-02-19 14:30:44'::timestamp without time zone) AND (date <= '2023-02-19
14:55:44'::timestamp without time zone))

 Planning Time: 0.199 ms

 Execution Time: 3.490 ms
```

BRIN Index results

2 GB table	BRIN	B-Tree
Index build time	11s	96s
Index size	24kB	1.1 GB
Load time w index	+30%	+200%
Index SEL (1 row)	X2-3	1
Index SEL (many)	same	same

Block Range Indexes

- Does not require complex DDL
- Constant overhead as table grows
- Generate almost no index inserts
  - Fits in RAM even for Petabytes of data
  - Generate almost no additional WAL
- Works well with Hot Standby data warehousing
- Additional indexing may be needed

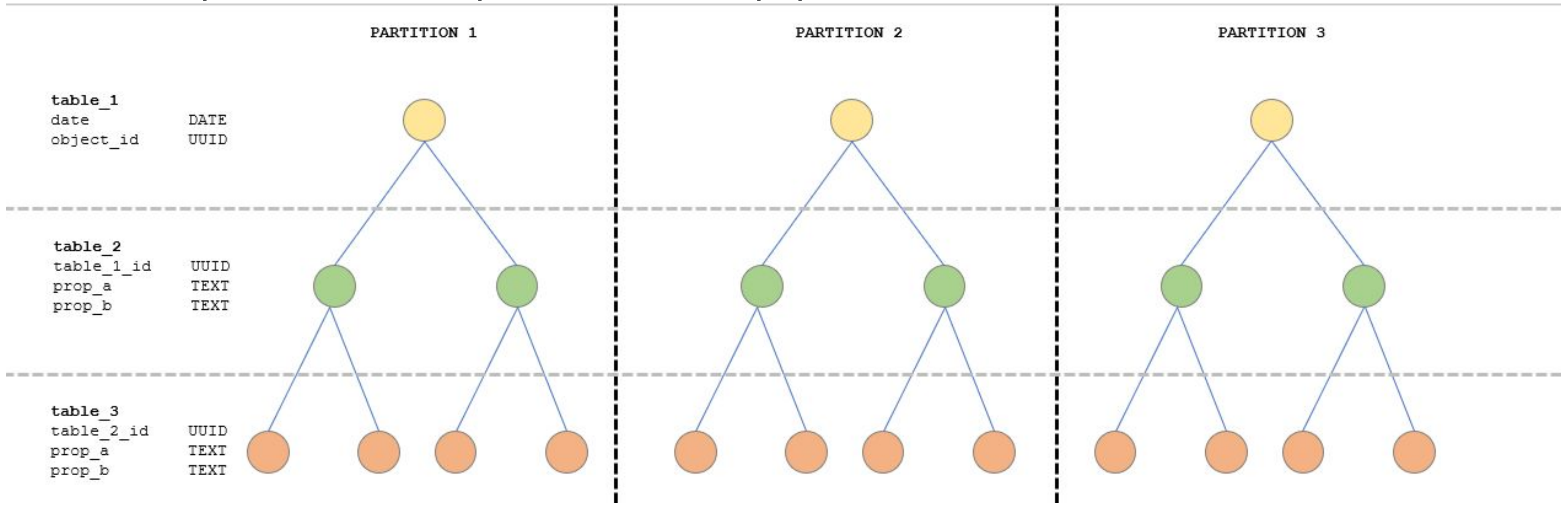
When to use partitioning?

Here are some suggestions for when to partition a table:

- Tables greater than 10 GB should always be considered as candidates for Partitioning.
- Tables containing historical data, in which new data is added into the newest partition. A typical example is a historical table where only the current month's data is updatable and the other 11 months are read only.
- When the contents of a table need to be distributed across different types of storage devices.

How to perform the partitioning?

- However, you want to perform deep partition



- Deeply segregating data by date will improve performance. Most heavy data query is done on table_3 so it would be nice to partition it by date which is determined by relationship from table_1.

Types of Partitioning

- **Range Partitioning :**

  The table is partitioned into “ranges” defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions.

- **List Partitioning :**

  The table is partitioned by explicitly listing which key value(s) appear in each partition.

- **Hash Partitioning :**

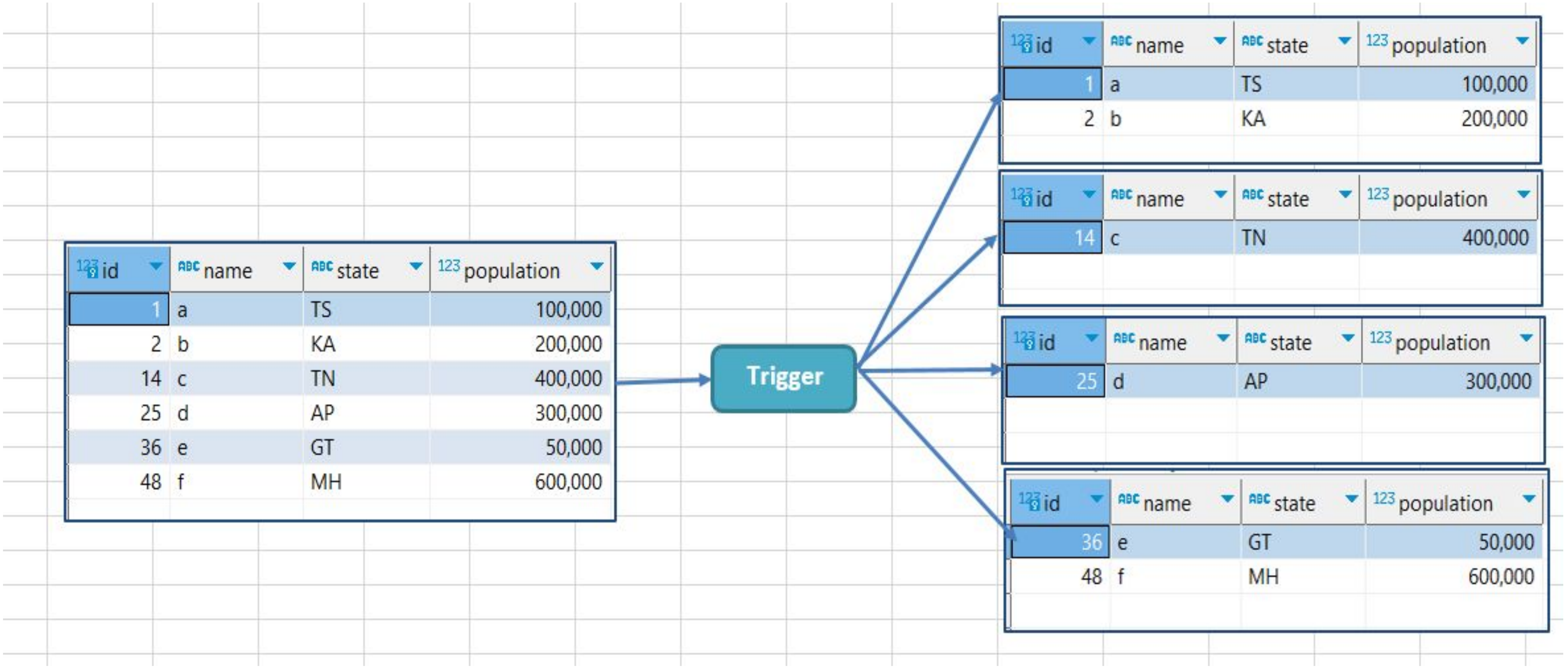
  The table is partitioned by specifying a modulus and a remainder for each partition. Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder.

Types of Partitioning

- **Composite Partition**

  To construct a more complex PostgreSQL partition layout, you can establish multiple partitions under a partition, as seen below. A Composite Partition, is sometimes known as a subpartition.

Range Partitioning



id	name	state	population
1	a	TS	100,000
2	b	KA	200,000
14	c	TN	400,000
25	d	AP	300,000
36	e	GT	50,000
48	f	MH	600,000

id	name	state	population
1	a	TS	100,000
2	b	KA	200,000

id	name	state	population
14	c	TN	400,000

id	name	state	population
25	d	AP	300,000

id	name	state	population
36	e	GT	50,000
48	f	MH	600,000

Range Partitioning - contd.

For example: Creating a table for range partitioning with 4 partition

```
CREATE TABLE city (
  id int4 NOT NULL PRIMARY KEY,
  name varchar(30) NOT NULL,
  state varchar(20),
  population int4
)PARTITION BY RANGE (id);

CREATE TABLE city_id1 PARTITION OF city FOR VALUES FROM (MINVALUE) TO (10);
CREATE TABLE city_id2 PARTITION OF city FOR VALUES FROM (10) TO (20);
CREATE TABLE city_id3 PARTITION OF city FOR VALUES FROM (20) TO (30);
CREATE TABLE city_id4 PARTITION OF city FOR VALUES FROM (30) TO (MAXVALUE);
```

Range Partitioning - cont.

```
INSERT INTO city values (1,'a','TS',100000);
INSERT INTO city values (2,'b','KA',200000);
INSERT INTO city values (14,'c','TN',400000);
INSERT INTO city values (25,'d','AP',300000);
INSERT INTO city values (36,'e','GT',50000);
INSERT INTO city values (48,'f','MH',600000);
```

Range Partitioning - contd.

```
select * from city;
 id | name | state | population
----+------+-------+------------
  1 | a    | TS    |     100000
  2 | b    | KA    |     200000
 14 | c    | TN    |     400000
 25 | d    | AP    |     300000
 36 | e    | GT    |      50000
 48 | f    | MH    |     600000
(6 rows)
```

Range Partitioning - contd.

```
select * from city_id1;
 id | name | state | population
----+------+-------+------------
  1 | a    | TS    |     100000
  2 | b    | KA    |     200000
select * from city_id2;
 id | name | state | population
----+------+-------+------------
 14 | c    | TN    |     400000

select * from city_id3;
 id | name | state | population
----+------+-------+------------
 25 | d    | AP    |     300000

select * from city_id4;
 id | name | state | population
----+------+-------+------------
 36 | e    | GT    |      50000
 48 | f    | MH    |     600000
```

List Partitioning

```
CREATE TABLE cities (
  city_id bigserial NOT NULL,
  name text NOT NULL,
  population bigint
)PARTITION BY LIST (left(lower(name), 1));

CREATE TABLE cities_ab PARTITION OF cities FOR VALUES IN ('a', 'b');
CREATE TABLE cities_cd PARTITION OF cities FOR VALUES IN ('c', 'd');
CREATE TABLE cities_ef PARTITION OF cities FOR VALUES IN ('e', 'f');
CREATE TABLE cities_gh PARTITION OF cities FOR VALUES IN ('g', 'h');
```

List Partitioning - contd.

```
INSERT INTO cities values (1, 'hyd', 1000000);
INSERT INTO cities values (2, 'abc', 1000000);
INSERT INTO cities values (3, 'bca', 2000000);
INSERT INTO cities values (4, 'cba', 3000000);
INSERT INTO cities values (5, 'boy', 3000000);
INSERT INTO cities values (7, 'crros', 4000000);
INSERT INTO cities values (8, 'drop', 4000000);
INSERT INTO cities values (9, 'egg', 4000000);
INSERT INTO cities values (10, 'forst', 5000000);
```

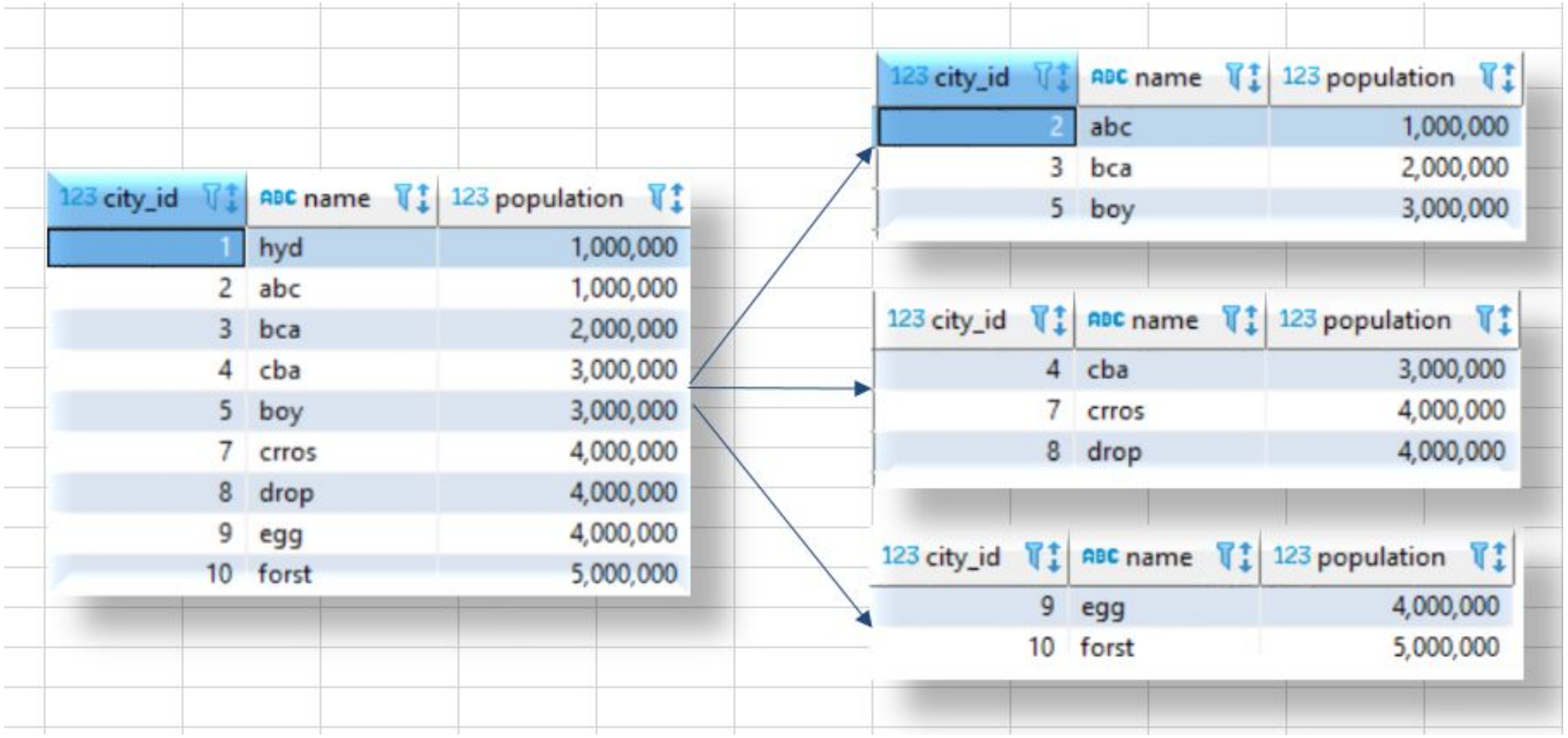
List Partitioning - contd.

```
postgres=# select * from cities_ab;
 city_id | name | population
---------+------+------------
       2 | abc  |    1000000
       3 | bca  |    2000000
       5 | boy  |    3000000

postgres=# select * from cities_cd;
 city_id | name   | population
---------+--------+------------
       4 | cba    |    3000000
       7 | crros  |    4000000
       8 | drop   |    4000000
(3 rows)
```

List Partitioning - contd.

```
postgres=# select * from cities_ef;
 city_id | name   | population
---------+--------+------------
       9 | egg    |    4000000
      10 | forst  |    5000000
(2 rows)

postgres=# select * from cities_gh;
 city_id | name | population
---------+------+------------
       1 | hyd  |    1000000
(1 row)
```

List Partitioning - contd.



city_id	name	population
1	hyd	1,000,000
2	abc	1,000,000
3	bca	2,000,000
4	cba	3,000,000
5	boy	3,000,000
7	crros	4,000,000
8	drop	4,000,000
9	egg	4,000,000
10	forst	5,000,000

city_id	name	population
2	abc	1,000,000
3	bca	2,000,000
5	boy	3,000,000

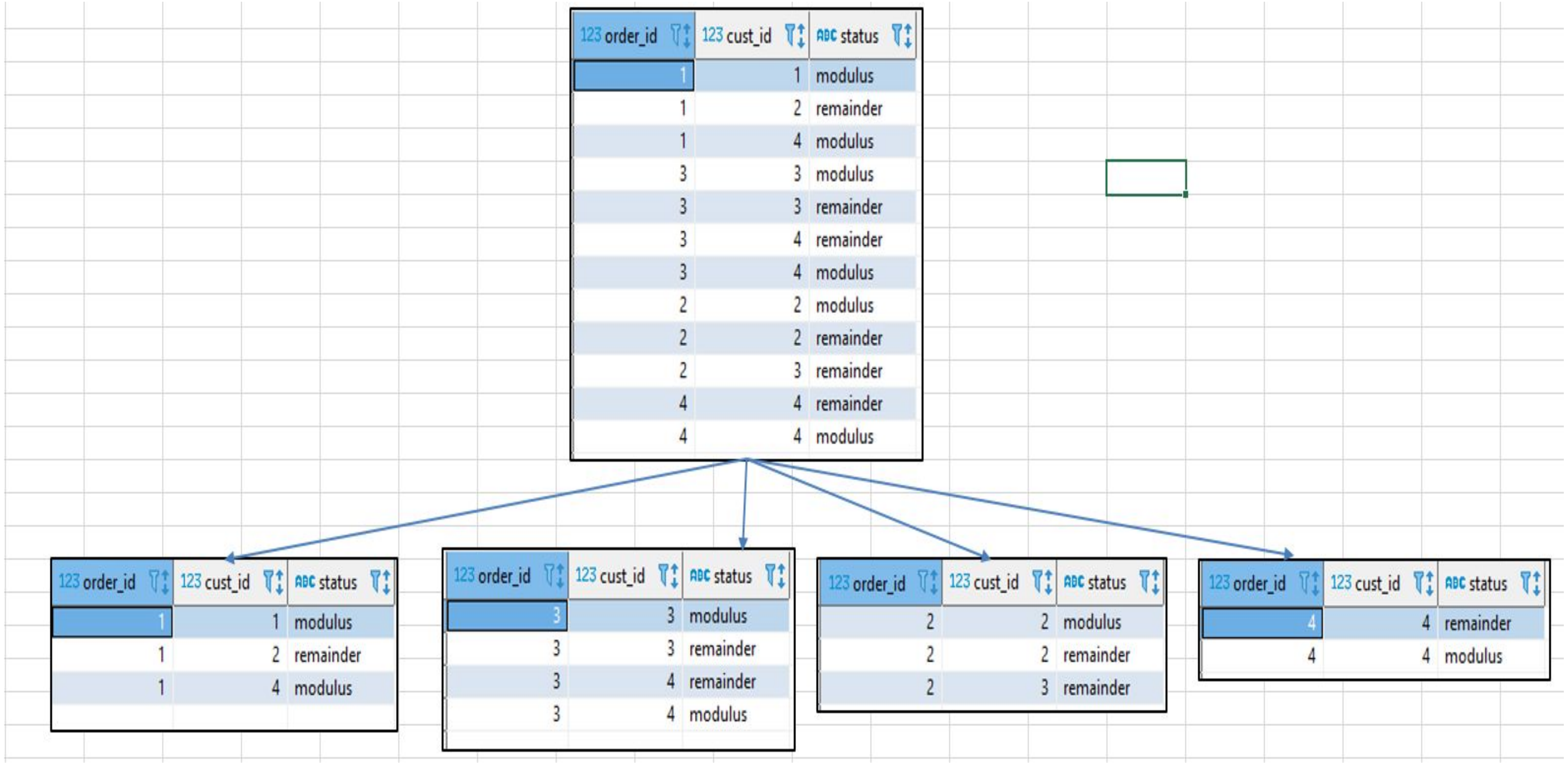
city_id	name	population
4	cba	3,000,000
7	crros	4,000,000
8	drop	4,000,000

city_id	name	population
9	egg	4,000,000
10	forst	5,000,000

Hash Partitioning

- The table is partitioned by specifying a modulus and a remainder for each partition
- Rather than group similar data as it does in range partitioning
- Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder
- Need to manually create all child tables
- Hash partitioning can not have a default partition as that would not make any sense because of the modulus and the remainder

Hash Partitioning - contd.



order_id	cust_id	status
1	1	modulus
1	2	remainder
1	4	modulus
3	3	modulus
3	3	remainder
3	4	remainder
3	4	modulus
2	2	modulus
2	2	remainder
2	3	remainder
4	4	remainder
4	4	modulus

order_id	cust_id	status
1	1	modulus
1	2	remainder
1	4	modulus

order_id	cust_id	status
3	3	modulus
3	3	remainder
3	4	remainder
3	4	modulus

order_id	cust_id	status
2	2	modulus
2	2	remainder
2	3	remainder

order_id	cust_id	status
4	4	remainder
4	4	modulus

Hash Partitioning - contd.

```
postgres=# CREATE TABLE orders (
  order_id bigint NOT NULL,
  cust_id bigint NOT NULL,
  status text
) PARTITION BY HASH (order_id);

CREATE TABLE orders_p1 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Hash Partitioning - contd.

```
INSERT INTO orders values (1, 1, 'modulus');
INSERT INTO orders values (2, 2, 'modulus');
INSERT INTO orders values (3, 3, 'modulus');
INSERT INTO orders values (1, 2, 'remainder');
INSERT INTO orders values (2, 2, 'remainder');
INSERT INTO orders values (2, 3, 'remainder');
INSERT INTO orders values (3, 3, 'remainder');
INSERT INTO orders values (3, 4, 'remainder');
INSERT INTO orders values (4, 4, 'remainder');
INSERT INTO orders values (4, 4, 'modulus');
INSERT INTO orders values (3, 4, 'modulus');
INSERT INTO orders values (1, 4, 'modulus');
```

Hash Partitioning - contd.

```
postgres=# select * from orders;
 order_id | cust_id |  status
----------+---------+-----------
        1 |       1 | modulus
        1 |       2 | remainder
        3 |       3 | modulus
        3 |       3 | remainder
        2 |       2 | modulus
        2 |       2 | remainder
        2 |       3 | remainder
(7 rows)
```

Hash Partitioning - contd.

```
postgres=# select * from orders_p1;
 order_id | cust_id |  status
----------+---------+-----------
        1 |       1 | modulus
        1 |       2 | remainder
(2 rows)
```

```
postgres=# select * from orders_p2;
 order_id | cust_id |  status
----------+---------+-----------
        3 |       3 | modulus
        3 |       3 | remainder
(2 rows)
```

```
postgres=# select * from orders_p3;
 order_id | cust_id |  status
----------+---------+-----------
        2 |       2 | modulus
        2 |       2 | remainder
        2 |       3 | remainder
(3 rows)
```

```
postgres=# select * from orders_p4;
 order_id | cust_id | status
----------+---------+--------
(0 rows)
```

Types of Partitioning

- **Composite Partition**

  To construct a more complex PostgreSQL partition layout, you can establish multiple partitions under a partition, as seen below. A Composite Partition, is sometimes known as a subpartition.

Type of Sub Partitions:

- List-List
- List-Range
- Range-Range
- Range-List

Types of Partitioning - Sub Partitions

List-List:

Let us understand how we can create table using list - list sub partitioning. We would like to have main partition per year and then sub partitions per quarter.

- Create table users_qtly with PARTITION BY LIST with created_year.
- Create tables for yearly partitions with PARTITION BY LIST with created_month.
- Create tables for quarterly partitions with list of values using FOR VALUES IN.

Types of Partitioning - Sub Partitions

List-List - Example:

```
CREATE TABLE sales (
    sale_id SERIAL PRIMARY KEY,
    sale_date DATE,
    sale_amount NUMERIC(10,2),
    region TEXT
)
PARTITION BY LIST(region)
SUBPARTITION BY LIST(sale_date)
SUBPARTITION TEMPLATE (
    SUBPARTITION q1 VALUES ('2023-01-01' to '2023-03-31'),
    SUBPARTITION q2 VALUES ('2023-04-01' to '2023-06-30'),
    SUBPARTITION q3 VALUES ('2023-07-01' to '2023-09-30'),
    SUBPARTITION q4 VALUES ('2023-10-01' to '2023-12-31'),
    DEFAULT SUBPARTITION other
);
```

Types of Partitioning - Sub Partitions

List-List - Example:

```
CREATE TABLE sales_usa PARTITION OF sales FOR VALUES IN ('usa');

CREATE TABLE sales_europe PARTITION OF sales FOR VALUES IN ('europe');

CREATE TABLE sales_q1_2023_usa PARTITION OF sales_usa  FOR VALUES IN ('2023-01-01' to '2023-03-31');

CREATE TABLE sales_q2_2023_usa PARTITION OF sales_usa FOR VALUES IN ('2023-04-01' to '2023-06-30');

CREATE TABLE sales_q3_2023_usa PARTITION OF sales_usa FOR VALUES IN ('2023-07-01' to '2023-09-30');

CREATE TABLE sales_q4_2023_usa PARTITION OF sales_usa FOR VALUES IN ('2023-10-01' to '2023-12-31');

CREATE TABLE sales_q1_2023_europe PARTITION OF sales_europe FOR VALUES IN ('2023-01-01' to
'2023-03-31');

CREATE TABLE sales_q2_2023_europe PARTITION OF sales_europe FOR VALUES IN ('2023-04-01' to
'2023-06-30');

CREATE TABLE sales_q3_2023_europe PARTITION OF sales_europe FOR VALUES IN ('2023-07-01' to
'2023-09-30');

CREATE TABLE sales_q4_2023_europe PARTITION OF sales_europe FOR VALUES IN ('2023-10-01' to
'2023-12-31');

CREATE TABLE sales_other PARTITION OF sales DEFAULT;
```

Types of Partitioning - Sub Partitions

List-Range:

Let us understand how we can create table using list - Range sub partitioning using same example as before (partitioning by year and then by quarter).

- Create table with PARTITION BY LIST with created_year.

- Create tables for yearly partitions with PARTITION BY RANGE with created_month.

- Create tables for quarterly partitions with the range of values using FOR VALUES FROM (lower_bound) TO (upper_bound).

Types of Partitioning - Sub Partitions

List-Range - Example:

```
CREATE TABLE sales (
    sale_id SERIAL PRIMARY KEY,
    sale_date DATE,
    sale_amount NUMERIC(10,2),
    region TEXT
)
PARTITION BY LIST(region)
SUBPARTITION BY RANGE(sale_date)
SUBPARTITION TEMPLATE (
    SUBPARTITION q1 VALUES LESS THAN ('2023-04-01'),
    SUBPARTITION q2 VALUES LESS THAN ('2023-07-01'),
    SUBPARTITION q3 VALUES LESS THAN ('2023-10-01'),
    SUBPARTITION q4 VALUES LESS THAN ('2024-01-01'),
    DEFAULT SUBPARTITION other
);
```

Types of Partitioning - Sub Partitions

List-Range - Example:

```
CREATE TABLE sales_usa PARTITION OF sales FOR VALUES IN ('usa');
CREATE TABLE sales_europe PARTITION OF sales FOR VALUES IN ('europe');
CREATE TABLE sales_q1_2023_usa PARTITION OF sales_usa FOR VALUES FROM ('2023-01-01') TO ('2023-04-01');
CREATE TABLE sales_q2_2023_usa PARTITION OF sales_usa FOR VALUES FROM ('2023-04-01') TO ('2023-07-01');
CREATE TABLE sales_q3_2023_usa PARTITION OF sales_usa FOR VALUES FROM ('2023-07-01') TO ('2023-10-01');
CREATE TABLE sales_q4_2023_usa PARTITION OF sales_usa FOR VALUES FROM ('2023-10-01') TO ('2024-01-01');
CREATE TABLE sales_q1_2023_europe PARTITION OF sales_europe FOR VALUES FROM ('2023-01-01') TO ('2023-04-01');
CREATE TABLE sales_q2_2023_europe PARTITION OF sales_europe  FOR VALUES FROM ('2023-04-01') TO ('2023-07-01');
CREATE TABLE sales_q3_2023_europe PARTITION OF sales_europe  FOR VALUES FROM ('2023-07-01') TO ('2023-10-01');
CREATE TABLE sales_q4_2023_europe PARTITION OF sales_europe FOR VALUES FROM ('2023-10-01') TO ('2024-01-01');
CREATE TABLE sales_other PARTITION OF sales
    DEFAULT;
```

Types of Partitioning - Sub Partitions

Range-Range:

- Range-range partitioning allows you to divide a large table into smaller, more manageable pieces based on both range. First the table is divided into the range partition and each partition is further divided into sub-partitions based on the range.

Types of Partitioning - Sub Partitions

Range-Range - Example:

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    order_date DATE,
    order_amount NUMERIC(10,2),
    customer_region TEXT,
    customer_age INTEGER
)
PARTITION BY RANGE (order_date, customer_age)
SUBPARTITION BY RANGE (order_amount)
SUBPARTITION TEMPLATE (
    SUBPARTITION low VALUES LESS THAN (100),
    SUBPARTITION medium VALUES LESS THAN (1000),
    SUBPARTITION high VALUES LESS THAN (10000),
    DEFAULT SUBPARTITION other
);
```

Types of Partitioning - Sub Partitions

Range-Range - Example:

```
CREATE TABLE orders_q1_2023_r1 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01', 18) TO ('2023-04-01', 25);
CREATE TABLE orders_q1_2023_r2 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01', 25) TO ('2023-04-01', 35);
CREATE TABLE orders_q1_2023_r3 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01', 35) TO ('2023-04-01', MAXVALUE);
CREATE TABLE orders_q2_2023_r1 PARTITION OF orders
    FOR VALUES FROM ('2023-04-01', 18) TO ('2023-07-01', 25);
CREATE TABLE orders_q2_2023_r2 PARTITION OF orders
    FOR VALUES FROM ('2023-04-01', 25) TO ('2023-07-01', 35);
CREATE TABLE orders_q2_2023_r3 PARTITION OF orders
    FOR VALUES FROM ('2023-04-01', 35) TO ('2023-07-01', MAXVALUE);
CREATE TABLE orders_other PARTITION OF orders
    DEFAULT;
```

Types of Partitioning - Sub Partitions

Range-List:

- Range-list partitioning is another partitioning method to partition a large table based on both range and a specified list of values. In range-list partitioning, each partition is created based on a range of values for a specific column, and a list of specific values for that same column.

Types of Partitioning - Sub Partitions

Range-List-Example:

```
CREATE TABLE sales (
    sale_id SERIAL PRIMARY KEY,
    sale_date DATE,
    sale_amount NUMERIC(10,2),
    region TEXT
)
PARTITION BY RANGE(sale_date)
SUBPARTITION BY LIST(region)
SUBPARTITION TEMPLATE (
    SUBPARTITION usa VALUES ('usa'),
    SUBPARTITION europe VALUES ('europe'),
    DEFAULT SUBPARTITION other
);
```

Types of Partitioning - Sub Partitions

```
CREATE TABLE sales_q1_2023 PARTITION OF sales
    FOR VALUES FROM ('2023-01-01') TO ('2023-04-01');
CREATE TABLE sales_q2_2023 PARTITION OF sales
    FOR VALUES FROM ('2023-04-01') TO ('2023-07-01');
-- subpartitions
CREATE TABLE sales_q1_2023_usa PARTITION OF sales_q1_2023
    FOR VALUES IN ('usa');
CREATE TABLE sales_q1_2023_europe PARTITION OF sales_q1_2023
    FOR VALUES IN ('europe');
CREATE TABLE sales_q1_2023_other PARTITION OF sales_q1_2023
    DEFAULT;
CREATE TABLE sales_q2_2023_usa PARTITION OF sales_q2_2023
    FOR VALUES IN ('usa');
CREATE TABLE sales_q2_2023_europe PARTITION OF sales_q2_2023
    FOR VALUES IN ('europe');
CREATE TABLE sales_q2_2023_other PARTITION OF sales_q2_2023
    DEFAULT;
```

Partitioning - Pros & Cons

- Manageability
- Fast Data Deletion and Data Load
- Piecemeal backup / restore of historical data
- Partition-wise index management
- Minimize index fragmentation for historically-partitioned tables
- Support alternative storage for historical data
- Performance querying Large Tables
- Smaller index tree or table scan when querying a single partition
- Queries may access partitions in parallel

Partitioning - Pros & Cons contd.

- To construct a unique or primary key constraint on a partitioned table, the partition keys cannot contain any expressions or function calls, and all partition key columns must be included in the constraint’s columns. Therefore, the partition structure must ensure that there are no duplicates between partitions.
- Temporary and permanent relations cannot coexist in the same partition tree.
- When utilizing temporary relations, all partition tree members must be from the same session.
- Partitioning could create constraints or operational burdens on the table that make using it difficult or intrusive to the code abstractions.
- Additional testing and code are usually required, since partitioned tables are not completely transparent to the application.
- If partitioning is not useful then it could actually be harmful. As one example, if a table is small or static or has too many buckets so that each partition is small, then partitioning badly could cause serious performance problems.

Partition Types - Matrix

Partitioning Method	PostgreSQL 15	PostgreSQL 14	PostgreSQL 13	PostgreSQL 12	PostgreSQL 11
Range	Yes	Yes	Yes	Yes	Yes
List	Yes	Yes	Yes	Yes	Yes
Hash	Yes	Yes	Yes	Yes	No
Reference	Yes	Yes	Yes	Yes	No
Composite	Yes	Yes	Yes	No	No
Subpartitioning	Yes	Yes	Yes	No	No

Partitioning Features in V-11

Major enhancements in PostgreSQL 11 include:

- Improvements to partitioning functionality, including:
  - Add support for partitioning by a hash key
  - Add support for PRIMARY KEY, FOREIGN KEY, indexes, and triggers on partitioned tables
  - Allow creation of a “default” partition for storing data that does not match any of the remaining partitions
  - UPDATE statements that change a partition key column now cause affected rows to be moved to the appropriate partitions
  - Improve SELECT performance through enhanced partition elimination strategies during query planning and execution

Partitioning Features in V-11 - contd.

- To overcome the some limitations of partitioning in version 10, new features added to postgres version 11,
- Possibility to define a default partition, to which any entry that wouldn’t fit a corresponding partition would be added to.
- `CREATE TABLE orders_default PARTITION OF orders DEFAULT;`
- Indexes added to the main table 'replicated' to the underlying partitions, which improved declarative partitioning usability.
- Support for Foreign Keys.
- Row movements across partitions update

Partitioning Features in V-11 - contd.

- Partitioning pruning - Improve select performance through enhanced partition elimination strategies during query planning and execution.
- Significantly cheaper plan than when enabled
- This is possible by parameter 'enable_partition_pruning' it can be set at session level.
  eg. set enable_partition_pruning= on/off
- Support Index, primary key and unique key constraints and triggers on partitioned table.
- Sub partitioning - adding table with partitions to previously created partitioned table, consider below example,
- Hash partition: In simple terms hash partitioning means dividing parent table into equal size of child tables,
- hash partitioning is useful for large tables containing no logical or natural value ranges to partition

Partitioning Features in V-12

- In Postgresql 12 the most noticeable enhancement is a performance improvement, there is a big focus on scaling partitioning to make it not only perform better, but perform better with a larger number of partitions
- Partitioning performance enhancements can improve query performance, particularly performance with INSERT and COPY statements.
- In Postgresql 12 users can have the ability to alter partitioned tables without blocking queries.
- Foreign Key references for partitioned tables.
- Add psql command \dP to list partitioned tables and indexes
- Improve psql \d and \z display of partitioned tables

Partitioning Features in V-13

- Allow logical replication into partitioned tables on subscribers
  - Previously, subscribers could only receive rows into non-partitioned tables.
- Allow whole-row variables (that is, table.*) to be used in partitioning expressions
- When using LOCK TABLE on a partitioned table, do not check permissions on the child tables
- Allow pgbench to partition its “accounts” table
  - This allows performance testing of partitioning.

Partitioning Features in V-13 - contd.

- Improved performance for queries that use aggregates or partitioned tables
- Allow pruning of partitions and partition wise joins to happen in more cases
- Allow partitioned tables to be logically replicated via publications
- Previously, partitions had to be replicated individually. Now a partitioned table can be published explicitly, causing all its partitions to be published automatically. Addition/removal of a partition causes it to be likewise added to or removed from the publication. The CREATE PUBLICATION option publish_via_partition_root controls whether changes to partitions are published as their own changes or their parent's.
- Allow logical replication into partitioned tables on subscribers
- Previously, subscribers could only receive rows into non-partitioned tables.
- Allow whole-row variables to be used in partitioning expressions

Partitioning Features in V-14

- Improve the performance of updates and deletes on partitioned tables with many partitions
- This change greatly reduces the planner's overhead for such cases, and also allows updates/deletes on partitioned tables to use execution-time partition pruning.
- Allow partitions to be detached in a non-blocking manner
- The syntax is ALTER TABLE ... DETACH PARTITION ... CONCURRENTLY, and FINALIZE.
- Ignore COLLATE clauses in partition boundary values

Partitioning Features in V-14 - contd.

- In logical replication, avoid double transmission of a child table's data

  If a publication includes both child and parent tables, and has the **publish_via_partition_root** option set, subscribers uselessly initiated synchronization on both child and parent tables. Ensure that only the parent table is synchronized in such cases.
- Previously any such clause had to match the collation of the partition key; but it's more consistent to consider that it's automatically coerced to the collation of the partition key.
- Allow REINDEX to process all child tables or indexes of a partitioned relation

Partitioning Features in V-14 - contd.

- Allow REINDEX to process all child tables or indexes of a partitioned relation
- Allow postgres_fdw to import table partitions if specified by **IMPORT FOREIGN SCHEMA ... LIMIT TO**

  By default, only the root of a partitioned table is imported.

- Fix construction of per-partition foreign key constraints while doing **ALTER TABLE ATTACH PARTITION**

  Previously, incorrect or duplicate constraints could be constructed for the newly-added partition.

Partitioning Features in V-15

- Improve planning time for queries referencing partitioned tables

  This change helps when only a few of many partitions are relevant.

- Allow ordered scans of partitions to avoid sorting in more cases

  Previously, a partitioned table with a DEFAULT partition or a LIST partition containing multiple values could not be used for ordered partition scans. Now they can be used if such partitions are pruned during planning.

- Fix **ALTER TRIGGER RENAME** on partitioned tables to properly rename triggers on all partitions
- Allow **CLUSTER** on partitioned tables

Partitioning Features in V-15 - contd.

- Improve planning time for queries referencing partitioned tables

Example:

```
 CREATE TABLE sales (
   id SERIAL PRIMARY KEY,
   sale_date DATE,
   amount NUMERIC
)
PARTITION BY RANGE (sale_date);
```

Partitioning Features in V-15 - contd.

```
CREATE TABLE sales_2021_01 PARTITION OF sales FOR VALUES FROM ('2021-01-01')
TO ('2021-01-31');

CREATE TABLE sales_2021_02 PARTITION OF sales FOR VALUES FROM ('2021-02-01')
TO ('2021-02-28');

CREATE TABLE sales_2021_03 PARTITION OF sales FOR VALUES FROM ('2021-03-01')
TO ('2021-03-31');
```

```
CREATE INDEX sales_date_index ON sales (sale_date);
```

Partitioning Features in V-15 - contd.

Output (in previous versions):

```
EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM sales WHERE sale_date >=
'2021-01-01' AND sale_date <= '2021-01-31';

Append  (cost=0.00..4.03 rows=1 width=12) (actual time=0.000..0.000 rows=0 loops=1)

  ->  Seq Scan on sales  (cost=0.00..0.00 rows=1 width=12) (actual time=0.000..0.000
rows=0 loops=1)

 Filter: ((sale_date >= '2021-01-01'::date) AND (sale_date <= '2021-01-31'::date))

 ->  Seq Scan on sales_2021_01 sales  (cost=0.00..4.03 rows=1 width=12) (never
executed)

 Filter: ((sale_date >= '2021-01-01'::date) AND (sale_date <= '2021-01-31'::date))

Planning Time: 0.193 ms
```

Partitioning Features in V-15 - contd.

Output (in PostgreSQL 15):

```
EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM sales WHERE sale_date >=
'2021-01-01' AND sale_date <= '2021-01-31';

Append  (cost=0.00..4.03 rows=1 width=12) (actual time=0.000..0.000 rows=0
loops=1)

  ->  Seq Scan on sales  (cost=0.00..0.00 rows=1 width=12) (actual
time=0.000..0.000 rows=0 loops=1)

Filter: ((sale_date >= '2021-01-01'::date) AND (sale_date <= '2021-01-31'::date))

   ->  Seq Scan on sales_2021_01 sales  (cost=0.00..4.03 rows=1 width=12) (never
executed)

 Filter: ((sale_date >= '2021-01-01'::date) AND (sale_date <= '2021-01-31'::date))

Planning Time: 0.010 ms
```

Partitioning Features in V-15 - contd.

- In this example, we create a partitioned table called `sales` and partition it by the `sale_date` column. We then create some partitions for different time periods and create an index on the `sale_date` column. We then query the table for sales in January 2021 and use the `EXPLAIN` command to see the query plan and planning time.

Partitioning Features in V-15 - contd.

- Fix **ALTER TRIGGER RENAME** on partitioned tables to properly rename triggers on all partitions

  Example:

```
 CREATE TABLE sales (
    id SERIAL PRIMARY KEY,
    sale_date DATE,
    amount NUMERIC
)
PARTITION BY RANGE (sale_date);
```

Partitioning Features in V-15 - contd.

```
-- Create a trigger on the partitioned table
CREATE TRIGGER sales_trigger
BEFORE INSERT ON sales
FOR EACH ROW
EXECUTE FUNCTION sales_function();

-- Rename the trigger on the partitioned table
ALTER TRIGGER sales_trigger RENAME TO new_sales_trigger;
```

Partitioning Features in V-15 - contd.

```
-- Check that the trigger was renamed on all partitions
SELECT trigger_name, event_object_table
FROM information_schema.triggers
WHERE event_object_table = 'sales';
```

Output

```
new_sales_trigger | sales
new_sales_trigger_1 | sales_1
new_sales_trigger_2 | sales_2
```

Partitioning Features in V-15 - contd.

- Improve foreign key behavior of updates on partitioned tables that move rows between partitions
- Previously, such updates ran a delete action on the source partition and an insert action on the target partition.

Example:

- In previous versions of PostgreSQL, the update would have caused a delete action on the source partition and an insert action on the target partition, which could lead to issues with foreign keys and trigger functions.
- However, in PostgreSQL 15, an update action is run on the partition root, which provides cleaner semantics and avoids these issues.

Partitioning Features in V-15 - contd.

- Allow ordered scans of partitions to avoid sorting in more cases

  Previously, a partitioned table with a DEFAULT partition or a LIST partition containing multiple values could not be used for ordered partition scans. Now they can be used if such partitions are pruned during planning.

  Example

- Allow ordered scans of partitions to avoid sorting in more cases Previously, a partitioned table with a DEFAULT partition or a LIST partition containing multiple values could not be used for ordered partition scans. Now they can be used if such partitions are pruned during planning.

Partitioning Features in V-15 - contd.

- Improve foreign key behavior of updates on partitioned tables that move rows between partitions

  Previously, such updates ran a delete action on the source partition and an insert action on the target partition.

Example:

- In previous versions of PostgreSQL, the update would have caused a delete action on the source partition and an insert action on the target partition, which could lead to issues with foreign keys and trigger functions. However, in PostgreSQL 15, an update action is run on the partition root, which provides cleaner semantics and avoids these issues.

CLUSTER on partitioned tables in V-15

For example: Create a table named city and inserted random data into this.

```
partition=# \d+
                                          List of relations
 Schema |   Name   |        Type       |  Owner   | Persistence | Access method |    Size    |
Description
--------+----------+-------------------+----------+-------------+---------------+------------+--
-----------
 public | city     | partitioned table | postgres | permanent   |               | 0 bytes    |
 public | city_id1 | table             | postgres | permanent   | heap          | 8192 bytes |
 public | city_id2 | table             | postgres | permanent   | heap          | 8192 bytes |
 public | city_id3 | table             | postgres | permanent   | heap          | 0 bytes    |
 public | city_id4 | table             | postgres | permanent   | heap          | 8192 bytes |
(5 rows)
```

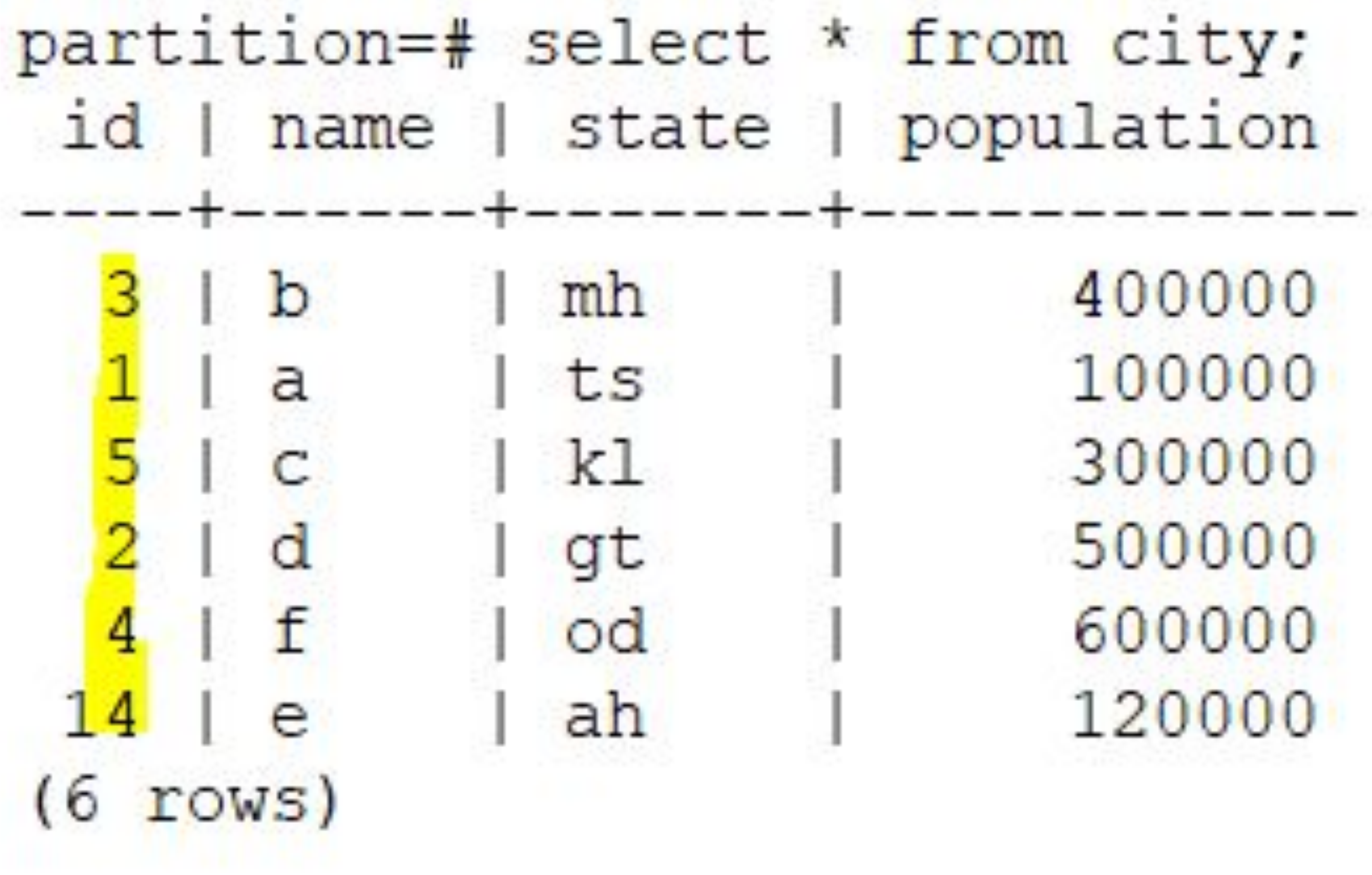
CLUSTER on partitioned tables in V-15 - contd.

```
partition=# \d+ city
                                              Partitioned table "public.city"
   Column   |         Type          | Collation | Nullable | Default | Storage  | Compression | Stats target | 
Description
------------+-----------------------+-----------+----------+---------+----------+-------------+--------------+-----------
---
 id         | integer               |           | not null |         | plain    |             |              |
 name       | character varying(30) |           | not null |         | extended |             |              |
 state      | character varying(20) |           |          |         | extended |             |              |
 population | integer               |           |          |         | plain    |             |              |
Partition key: RANGE (id)
Indexes:
    "city_pkey" PRIMARY KEY, btree (id)
    "idx_city" btree (id)
Partitions: city_id1 FOR VALUES FROM (MINVALUE) TO (10),
            city_id2 FOR VALUES FROM (10) TO (20),
            city_id3 FOR VALUES FROM (20) TO (30),
            city_id4 FOR VALUES FROM (30) TO (MAXVALUE)
```

CLUSTER on partitioned tables in V-15 - contd.

Before cluster command data in the city table as following:

```
partition=# select * from city;
 id | name | state | population
----+------+-------+------------
  3 | b    | mh    |     400000
  1 | a    | ts    |     100000
  5 | c    | kl    |     300000
  2 | d    | gt    |     500000
  4 | f    | od    |     600000
 14 | e    | ah    |     120000
(6 rows)
```

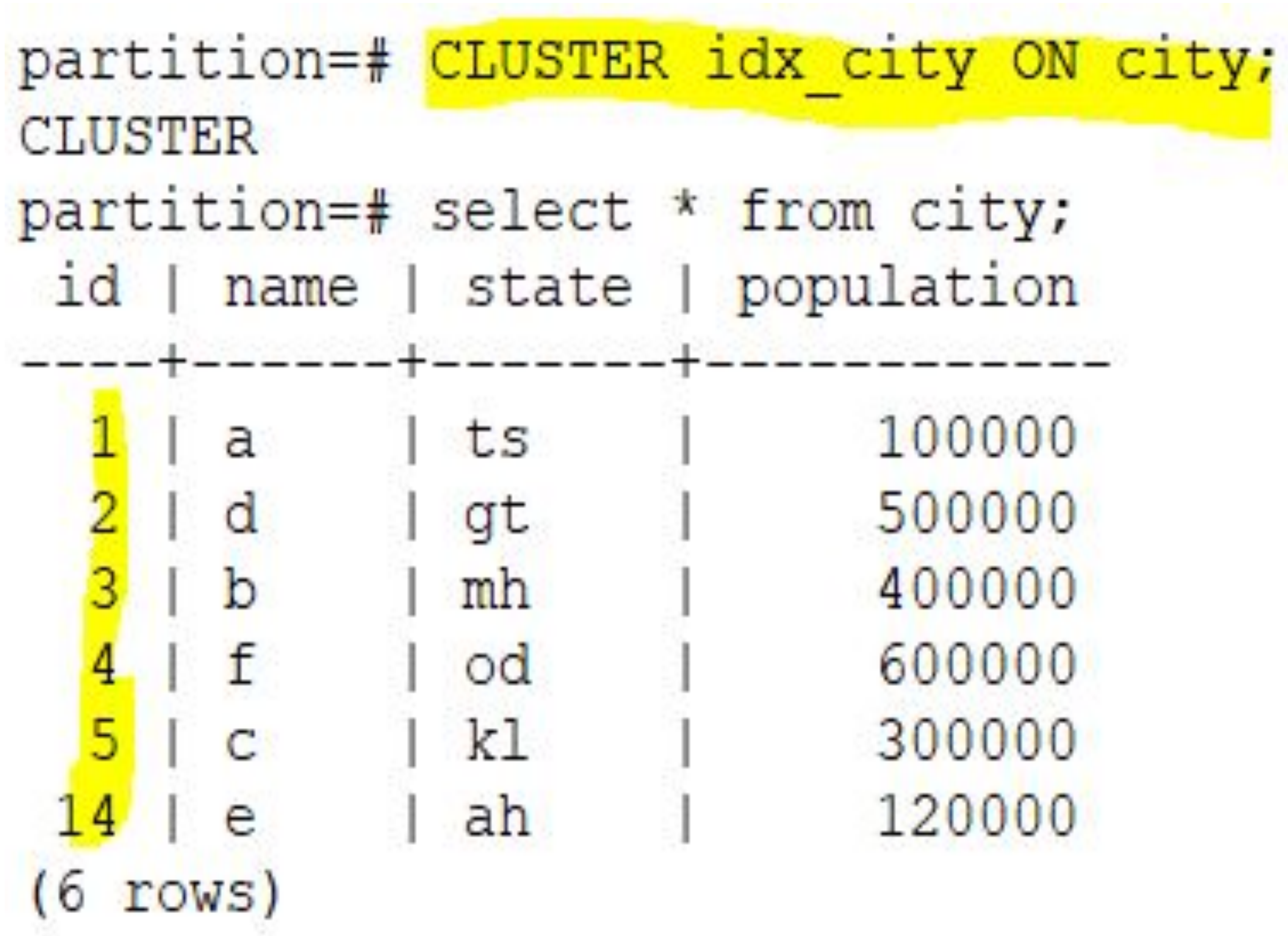


```
partition=# select * from city;
 id | name  | state  | population
----+-------+--------+------------
  3 | b     | mh     |     400000
  1 | a     | ts     |     100000
  5 | c     | kl     |     300000
  2 | d     | gt     |     500000
  4 | f     | od     |     600000
 14 | e     | ah     |     120000
(6 rows)
```

CLUSTER on partitioned tables in V-15 - contd.

After `CLUSTER` command in the city table data as following

```
partition=# CLUSTER idx_city ON city;
CLUSTER
partition=# select * from city;
 id | name | state | population
----+------+-------+-----------
  1 | a    | ts    |     100000
  2 | d    | gt    |     500000
  3 | b    | mh    |     400000
  4 | f    | od    |     600000
  5 | c    | kl    |     300000
 14 | e    | ah    |     120000
(6 rows)
```



```
partition=# CLUSTER idx_city ON city;
CLUSTER
partition=# select * from city;
 id | name | state | population
----+------+-------+-----------
  1 | a    | ts    |     100000
  2 | d    | gt    |     500000
  3 | b    | mh    |     400000
  4 | f    | od    |     600000
  5 | c    | kl    |     300000
 14 | e    | ah    |     120000
(6 rows)
```

Partitioning Features in V-15 - contd.

- Fix calculation of which GENERATED columns need to be updated in child tables during an UPDATE on a partitioned table or inheritance tree.

Example:

- Consider a partitioned table for employee data with a parent table named "employee" and two child tables named "full_time_employee" and "part_time_employee", which inherit from the parent table. The parent table has a GENERATED column "salary" that depends on the "pay_rate" and "hours_worked" columns. The child tables also have additional GENERATED columns that depend on other columns.

Partitioning Features in V-15 - contd.

Assume we want to update the "pay_rate" column for all employees. Here is an example query:

```
CREATE TABLE employee1 (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  pay_rate NUMERIC NOT NULL,
  hours_worked NUMERIC NOT NULL,
  salary NUMERIC GENERATED ALWAYS AS (pay_rate * hours_worked) STORED
) PARTITION BY RANGE (id);
```

Partitioning Features in V-15 - contd.

```
CREATE TABLE full_time_employee1 PARTITION OF employee FOR VALUES FROM (0) TO
(100);

CREATE TABLE part_time_employee1 PARTITION OF employee FOR VALUES FROM (100) TO
(MAXVALUE);

UPDATE employee1 SET pay_rate = pay_rate * 1.05;
```

Output:

```
 insert into employee1 values (1,'OSDB', 1, 10);

 insert into employee1 values (2,'Rama', 1, 10);

 insert into employee1 values (3,'sudheer', 1, 10);

 insert into employee1 values (4,'srini', 2, 20);

insert into employee1 values (5,'hari', 3, 20);
```

Partitioning Features in V-15 - contd.

```
insert into employee1 values (6,'srinu', 4, 20);
insert into employee1 values (7,'srinu', 5, 30);
insert into employee1 values (8,'RC', 5, 30);

select * from employee1;
 id |   name    | pay_rate | hours_worked | salary
----+----------+----------+--------------+--------
  1 | OSDB     |        1 |           10 |     10
  2 | Rama     |        1 |           10 |     10
  3 | sudheer  |        1 |           10 |     10
  4 | srini    |        2 |           20 |     40
  5 | hari     |        3 |           20 |     60
  6 | srinu    |        4 |           20 |     80
  7 | srinu    |        5 |           30 |    150
  8 | RC       |        5 |           30 |    150
```

Partitioning Features in V-15 - contd.

```
UPDATE employee1 SET pay_rate = pay_rate * 1.05;
UPDATE 8
select * from employee1;
 id |  name    | pay_rate | hours_worked | salary
----+----------+----------+--------------+--------
  1 | OSDB     |     1.05 |           10 |  10.50
  2 | Rama     |     1.05 |           10 |  10.50
  3 | sudheer  |     1.05 |           10 |  10.50
  4 | srini    |     2.10 |           20 |  42.00
  5 | hari     |     3.15 |           20 |  63.00
  6 | srinu    |     4.20 |           20 |  84.00
  7 | srinu    |     5.25 |           30 | 157.50
  8 | RC       |     5.25 |           30 | 157.50
```

Partitioning Features in V-15 - contd.

- In previous versions of PostgreSQL, this query would fail to update the "salary" column in the child tables, since it has a different dependency than the parent column's generation expression. However, with the improvement in PostgreSQL 15, the query will correctly update the "salary" column in the child tables, as well as any other GENERATED columns with different dependencies than the parent column's generation expression.

- This improvement ensures that all GENERATED columns in the child tables are correctly updated during an UPDATE on the parent table, even if they have different dependencies than the parent column's generation expression.

Interesting additions in V-16

- \d+ <partition table> indicates FOREIGN in the relation description of partitions
  you can thank him in-person here :-)

- For RANGE and LIST partitioned tables - store the details of the cached partition in PartitionDesc (i.e. relcache) so that the cached values are maintained over multiple statements. Not available for HASH partitioning

DB Sharding

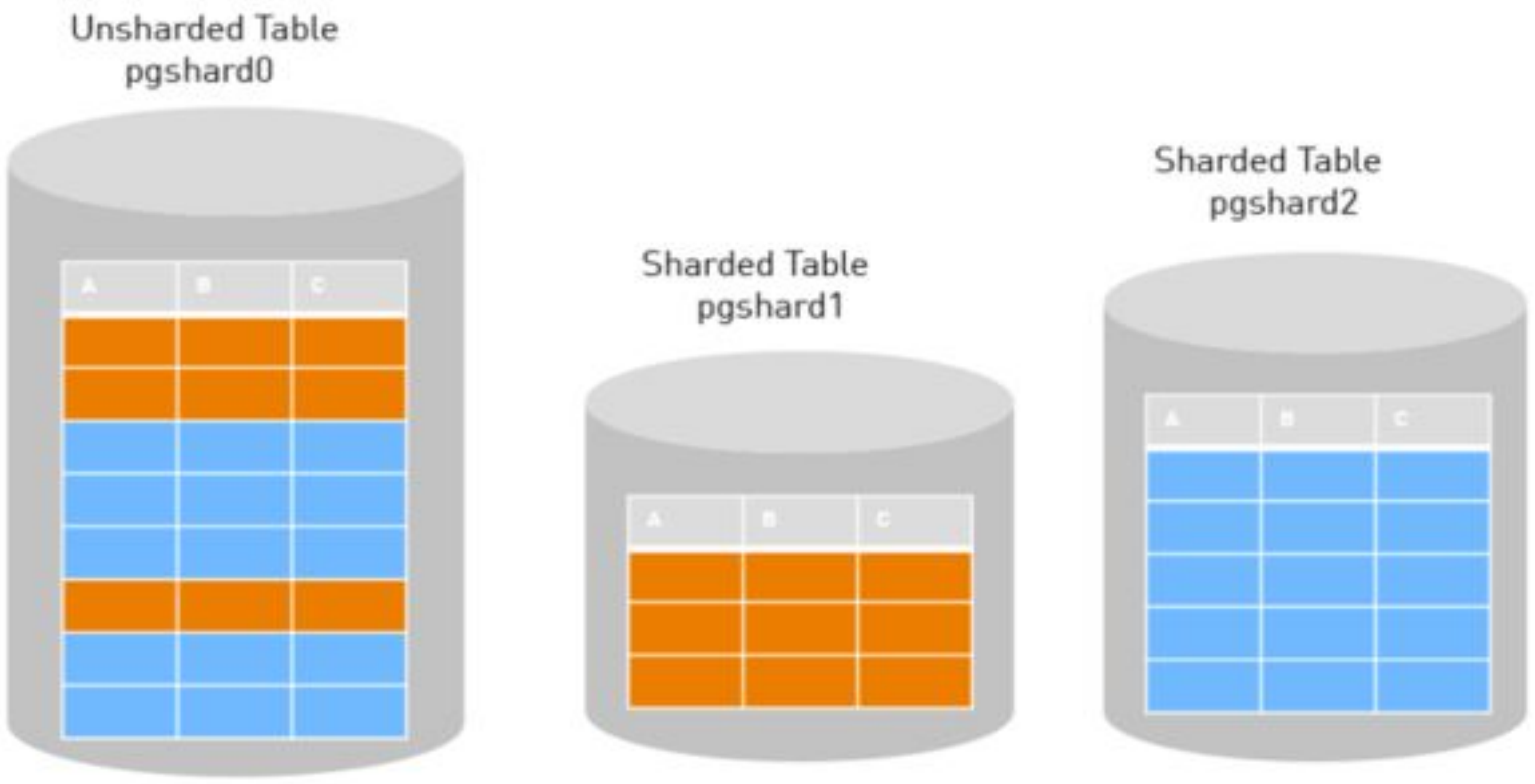
- What is Sharding?
- Why Is Sharding Used?
- How sharding works?
- Key differences between Partitioning and Sharding

What is Sharding?

Sharding is actually a type of database partitioning, more specifically, Horizontal Partitioning. Sharding, is replicating [copying] the schema, and then dividing the data based on a shard key onto a separate database server instance, to spread load.

- Sharding – Sharding is actually a type of database partitioning, Each of the partitions is located on a separate server. The new tables are called “shards”.

What is Sharding? - contd.



Why Is Sharding Used?

By sharding a larger table, you can store the new chunks of data, called **logical shards**, across multiple nodes to achieve horizontal scalability and improved performance.

Horizontal sharding is effective when queries tend to return a subset of rows that are often grouped together.

For example, queries that filter data based on short date ranges are ideal for horizontal sharding since the date range will necessarily limit querying to only a subset of the servers.

Vertical sharding is effective when queries tend to return only a subset of columns of the data.

For example, if some queries request only names, and others request only addresses, then the names and addresses can be sharded on to separate servers.

What Is Sharding? - contd.

Example:

Here is an example of how to shard a table called orders based on the order_date column

```
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    order_date DATE NOT NULL,
    customer_id INTEGER NOT NULL,
    ...
)
PARTITION BY RANGE (order_date);
```

What Is Sharding? - contd.

Insert data into partitions:

```
INSERT INTO orders_2021 (order_date, customer_id, ...) VALUES
('2021-01-01', 1, ...);
INSERT INTO orders_2022 (order_date, customer_id, ...) VALUES
('2022-01-01', 2, ...);
INSERT INTO orders_2023 (order_date, customer_id, ...) VALUES
('2023-01-01', 3, ...);
```

This inserts data into the appropriate partitions based on the order_date.

What Is Sharding? - contd.

Output:

```
postgres=# select * from sharding.orders;

 order_id | order_date | customer_id | customer_name
----------+------------+-------------+---------------
        1 | 2019-01-01 |           1 | Mac
        2 | 2020-01-01 |           2 | Zin
        3 | 2021-01-01 |           3 | Care
(3 rows)
```

This query retrieves data from all three partitions and combines the results into a single result set.

Create Sharding Tables

This creates an empty partitioned table and specifies that it should be partitioned by range, using the order_date column as the partition key.

```
CREATE TABLE orders_2021 PARTITION OF orders
    FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');
CREATE TABLE orders_2022 PARTITION OF orders
    FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');
CREATE TABLE orders_2023 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
This creates three partitions for the orders table, one for each year.
```

Citus database extension

- Citus is a distributed database built entirely as an open source extension for PostgreSQL.
- With Citus, you can create tables that are transparently distributed or replicated across a cluster of PostgreSQL nodes.
- Citus is available as open source and also as a managed service on Azure.
- Shard your Postgres tables without blocking writes
- Citus supports Postgres 15 GA—with improved sort performance & compression, expressive developer features, and more.
- Citus last release 11.2

When to Use Citus

Multi-Tenant SaaS Database

The database serves many tenants, each of whose data is separate from other tenants.

sharing the same database schema across multiple tenants makes efficient use of hardware resources and simplifies database management.

Advantages of Citus for multi-tenant applications:

- Fast queries for all tenants
- Sharding logic in the database, not the application
- Hold more data than possible in single-node PostgreSQL
- Scale out without giving up SQL
- Maintain performance under high concurrency
- Fast metrics analysis across customer base
- Easily scale to handle new customer signups
- Isolate resource usage of large and small customers

When to Use Citus - contd.

Real-Time Analytics

Citus supports real-time queries over large datasets. Commonly these queries occur in rapidly growing event systems or systems with time series data. Example use cases include:

- Analytic dashboards with subsecond response times
- Exploratory queries on unfolding events
- Large dataset archival and reporting
- Analyzing sessions with funnel, segmentation, and cohort queries

When to Use Citus - contd.

Citus’ benefits here are its ability to parallelize query execution and scale linearly with the number of worker databases in a cluster.

Some advantages of Citus for real-time applications:

- Maintain sub-second responses as the dataset grows
- Analyze new events and new data as it happens, in real-time
- Parallelize SQL queries
- Scale out without giving up SQL
- Maintain performance under high concurrency
- Fast responses to dashboard queries
- Use one database, not a patchwork
- Rich PostgreSQL data types and extensions

Citus database extension

Citus Extension :

```
Install the Citus extension:
CREATE EXTENSION citus;
```

Set up a coordinator node:

```
SELECT citus_add_node('coordinator_node', 5432);
```

This sets up a coordinator node that will act as the interface for your sharded database.

Set up worker nodes:

```
SELECT citus_add_node('worker_node1', 5432);
SELECT citus_add_node('worker_node2', 5432);
```

These commands add two worker nodes to the cluster, which will store the sharded data.

Citus database extension - contd.

Create a distributed table:

```
CREATE TABLE orders (
    order_id SERIAL,
    order_date DATE NOT NULL,
    customer_id INTEGER NOT NULL,
    ...
)
DISTRIBUTED BY (order_date);
```

This creates a distributed table that will be sharded across the worker nodes based on the order_date column.

Citus database extension - contd.

Insert data into the distributed table:

```
INSERT INTO orders (order_date, customer_id, ...) VALUES
('2019-01-01', 1, ...);

INSERT INTO orders (order_date, customer_id, ...) VALUES
('2020-01-01', 2, ...);

INSERT INTO orders (order_date, customer_id, ...) VALUES
('2021-01-01', 3, ...);
```

This inserts data into the distributed table, which will be automatically sharded across the worker nodes based on the order_date.

Citus for sharding provides automatic partitioning, rebalancing and query routing, which makes it a convenient and efficient solution for scaling a PostgreSQL database.

Key differences between Sharding & Partitioning?

Sharding and partitioning are both about breaking up a large data set into smaller subsets.

- Partitioning – Tables on single servers are divided into independent parts by some criteria. Partitioning is about grouping subsets of data within a single database instance.
- Sharding – Sharding is actually a type of database partitioning, for Horizontal scalability, across multiple database instances.

Combine the best of Both Worlds!

Git repo

https://github.com/opensource-db/pgconf-in-2023

Questions ?

OpenSource DB™

PostgreSQL MySQL Data Science

https://opensource-db.com/

THANK YOU!

Sudheer S

Senior DBA

sudheer.s@opensource-db.com

HARI P KIRAN

FOUNDER & PRINCIPAL

harikiran@opensource-db.com

RamaRao T

Senior DBA

ramarao.t@opensource-db.com