EDB POSTGRES

PGConf India

THE TRUTH ABOUT PARTITIONS

Beena Jolly Emerson

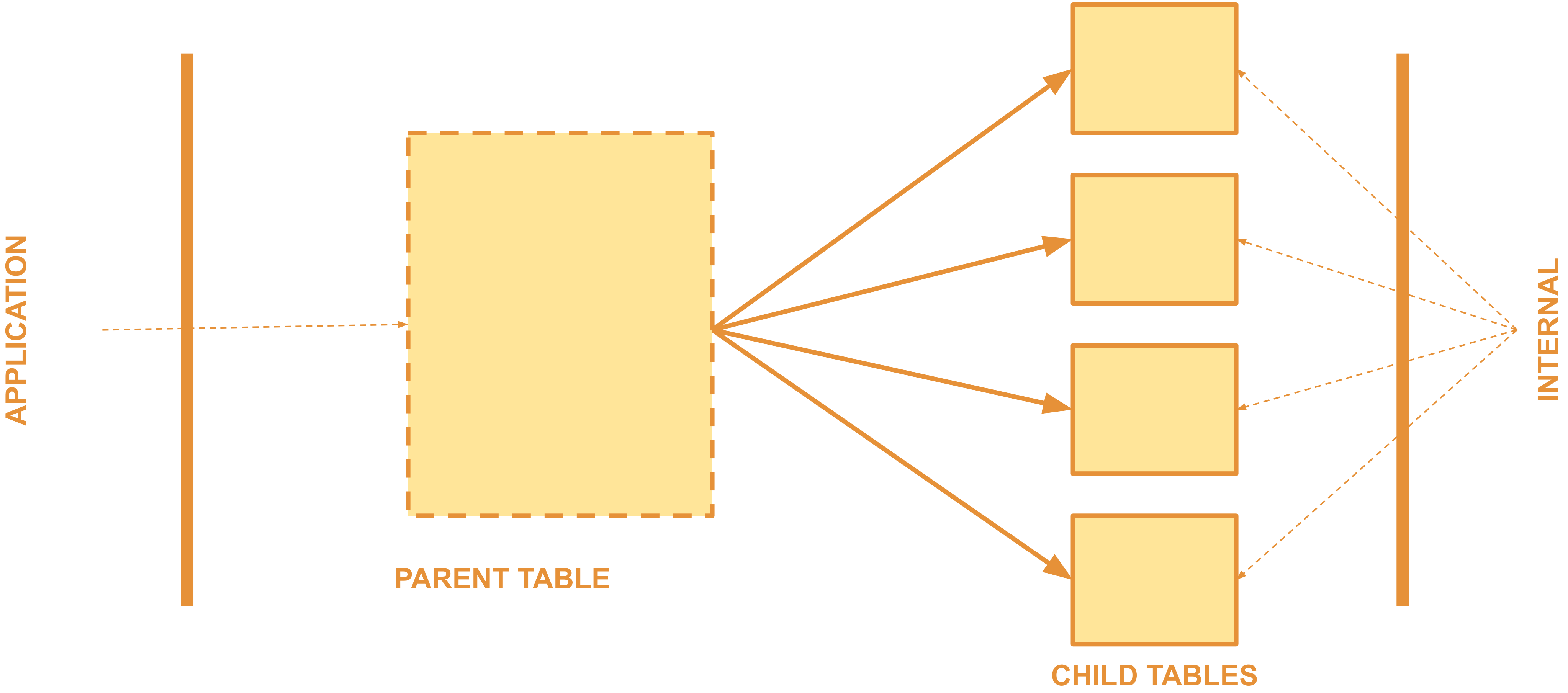
February 28, 2020

MBeenaEmerson

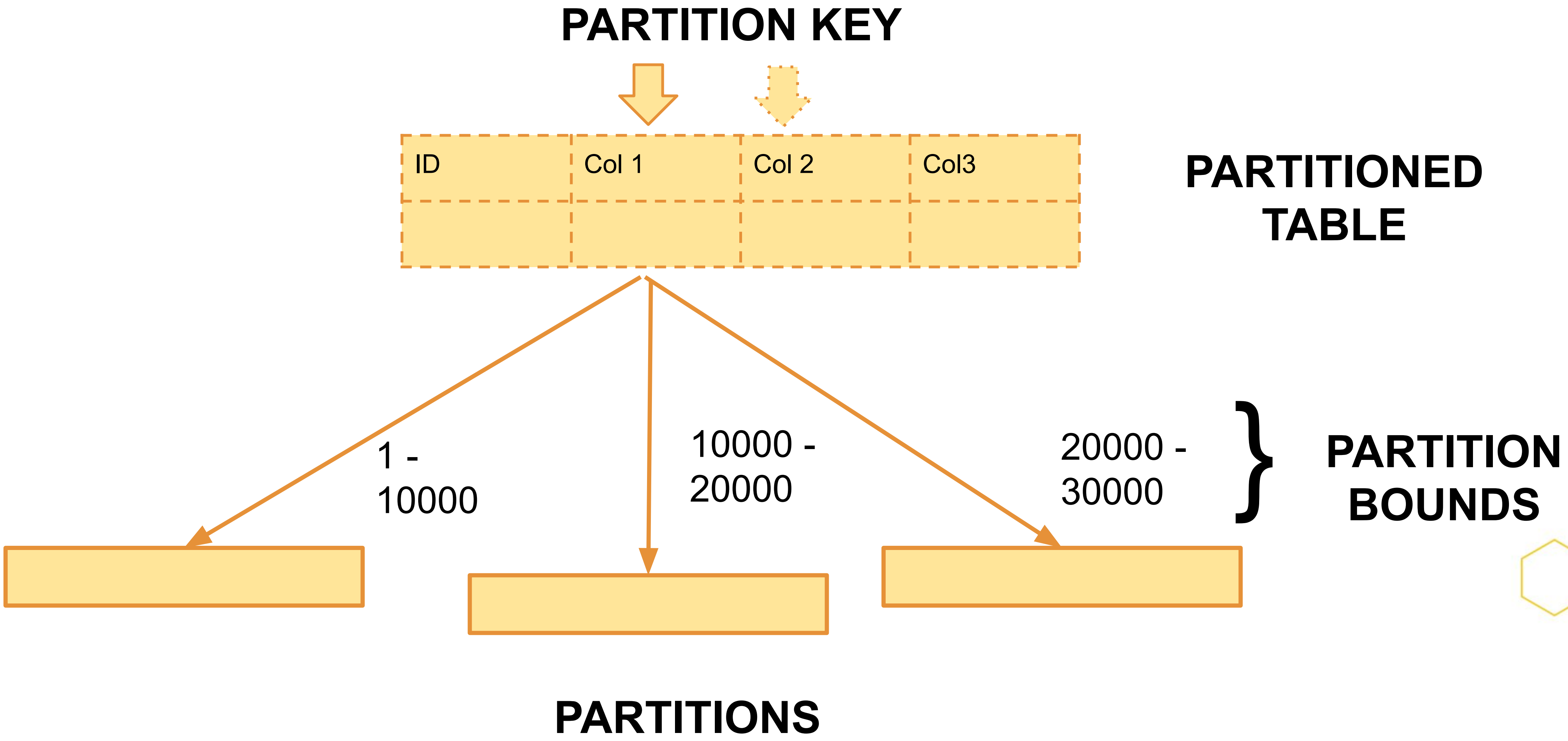
INTRODUCTION

- **What** is Partitioning?
- Key Terms
- Partitioning Benefits

What is Partitioning?

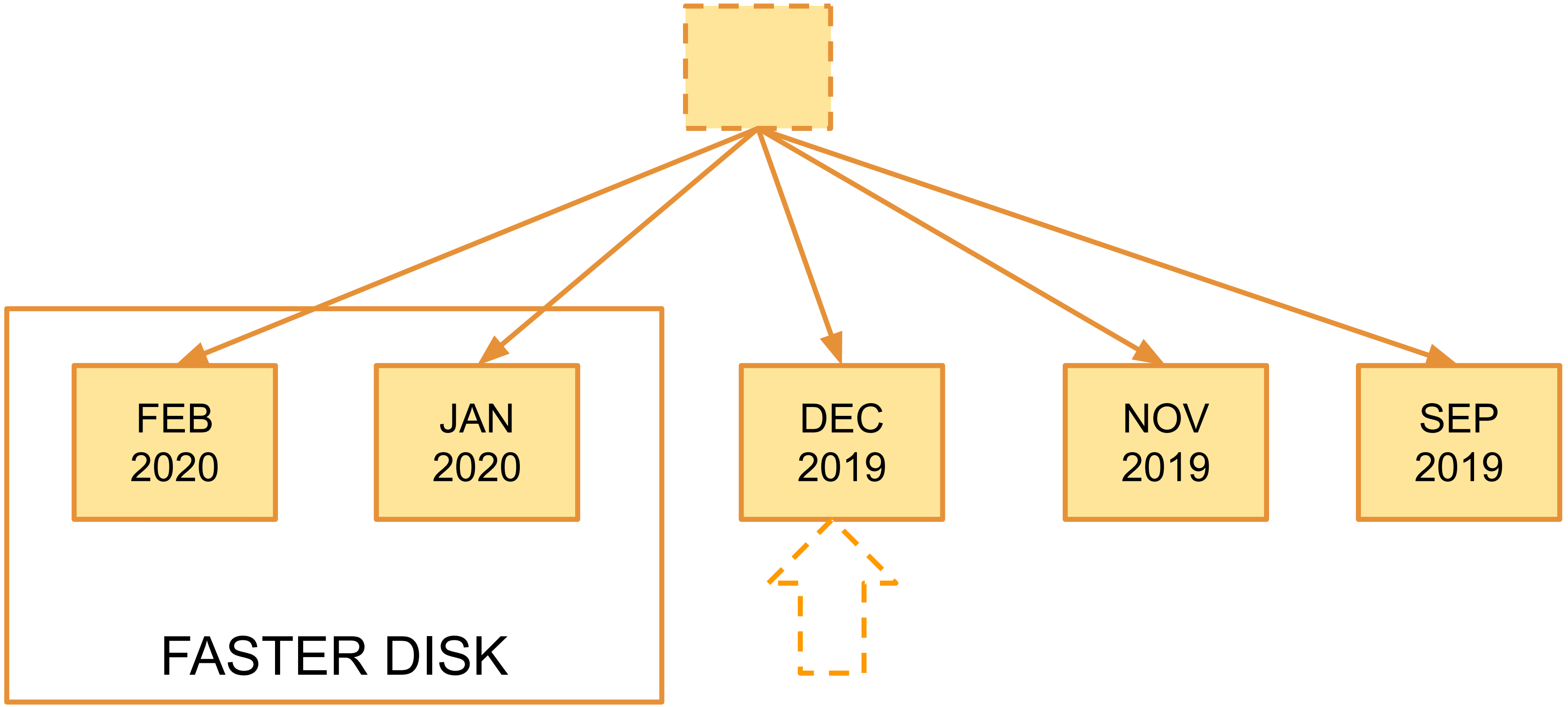


Key Terms


PARTITION KEY
ID
Col 1
Col 2
Col3
PARTITIONED TABLE
1 - 10000
10000 - 20000
20000 - 30000
PARTITION BOUNDS
PARTITIONS

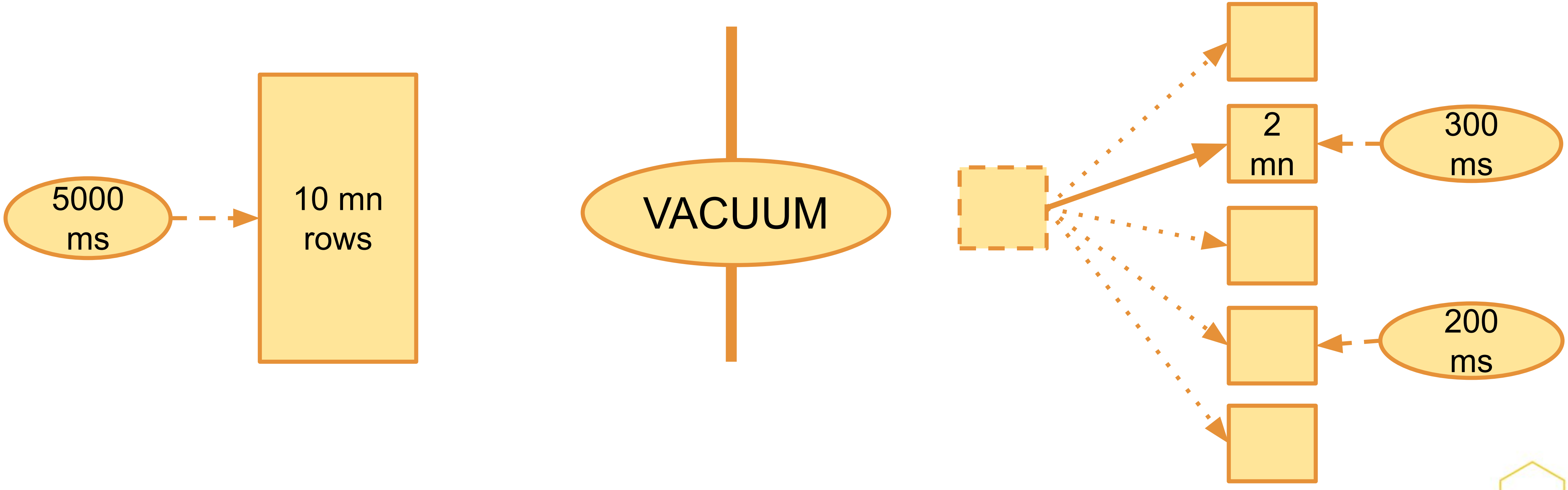
Partitioning Benefits

Data Segregation



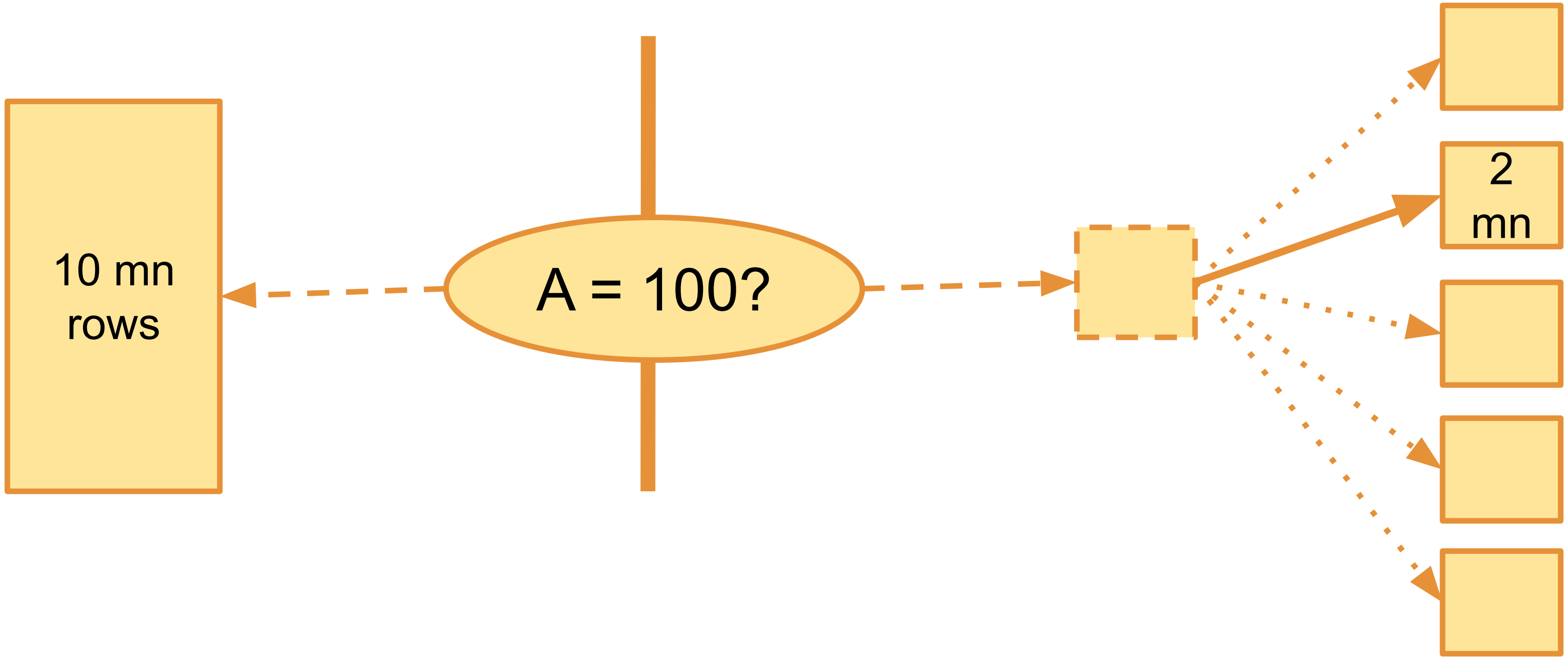
Partitioning Benefits

Maintenance



Partitioning Benefits

Performance / Scalability



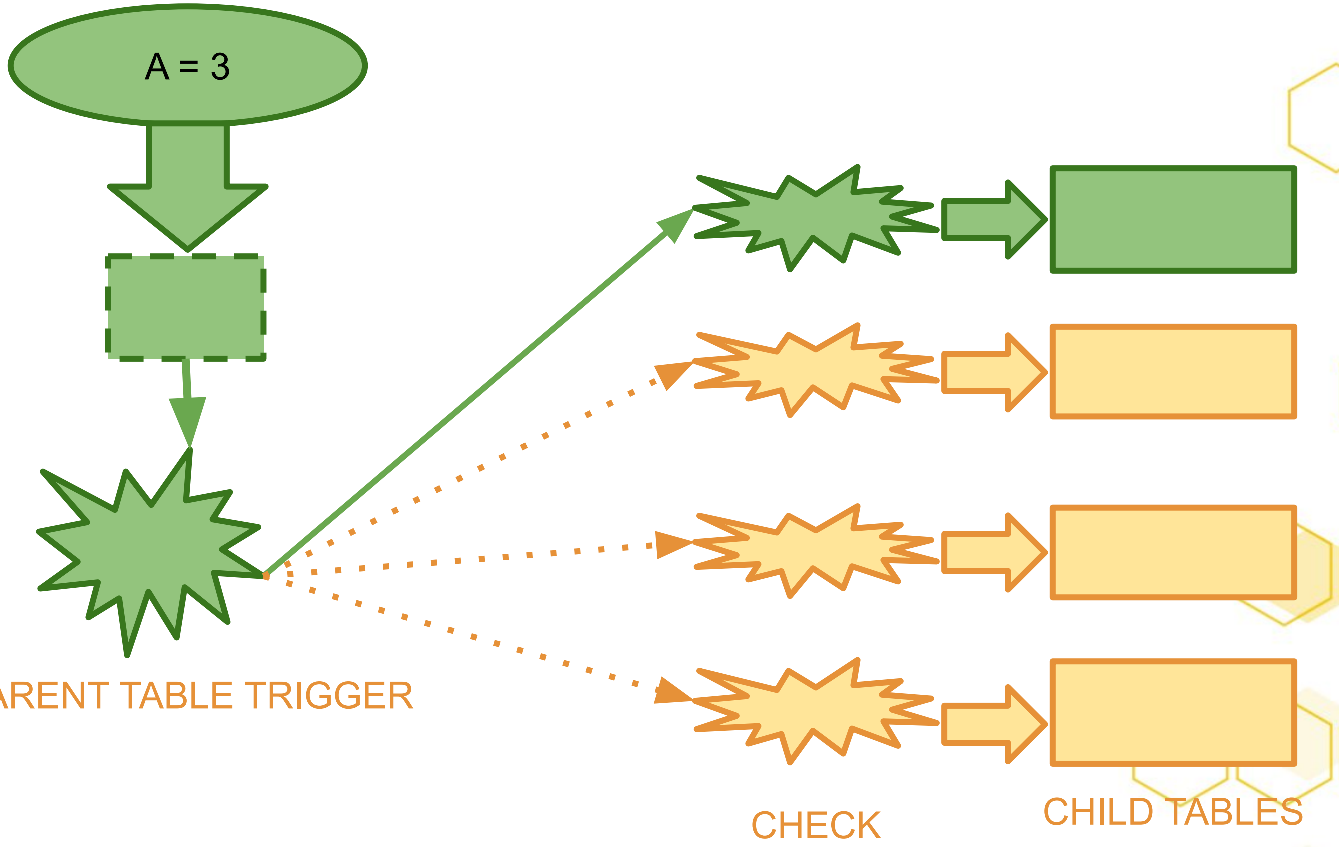
Scanning smaller tables can take less time.

STYLE OF PARTITIONING IN POSTGRESQL

- Inheritance (Trigger- based) Partition
- Declarative Partitioning

Inheritance (Trigger-Based) Partitions

- Manual
- Error-prone
- Constraints not mutually exclusive
- Hard to maintain partitions



Declarative Partitioning

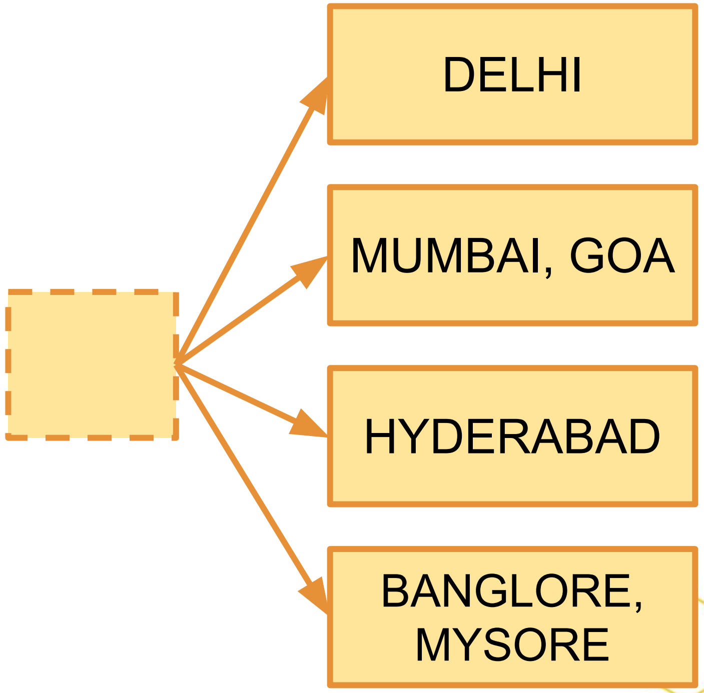
- PostgreSQL 10
- Automated - No manual handling of triggers.
- Simpler syntax
- Easy management of partitions

PARTITIONING STRATEGIES

- List
- Range
- Hash

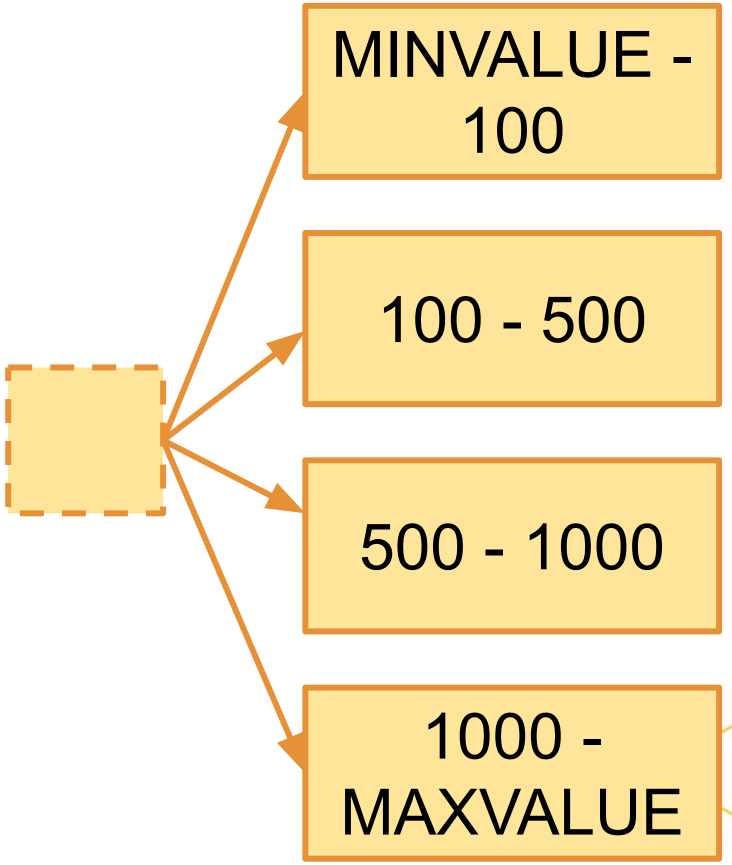
List Partitioning

- PostgreSQL 10
- Explicitly mention values for partition key - single or multiple



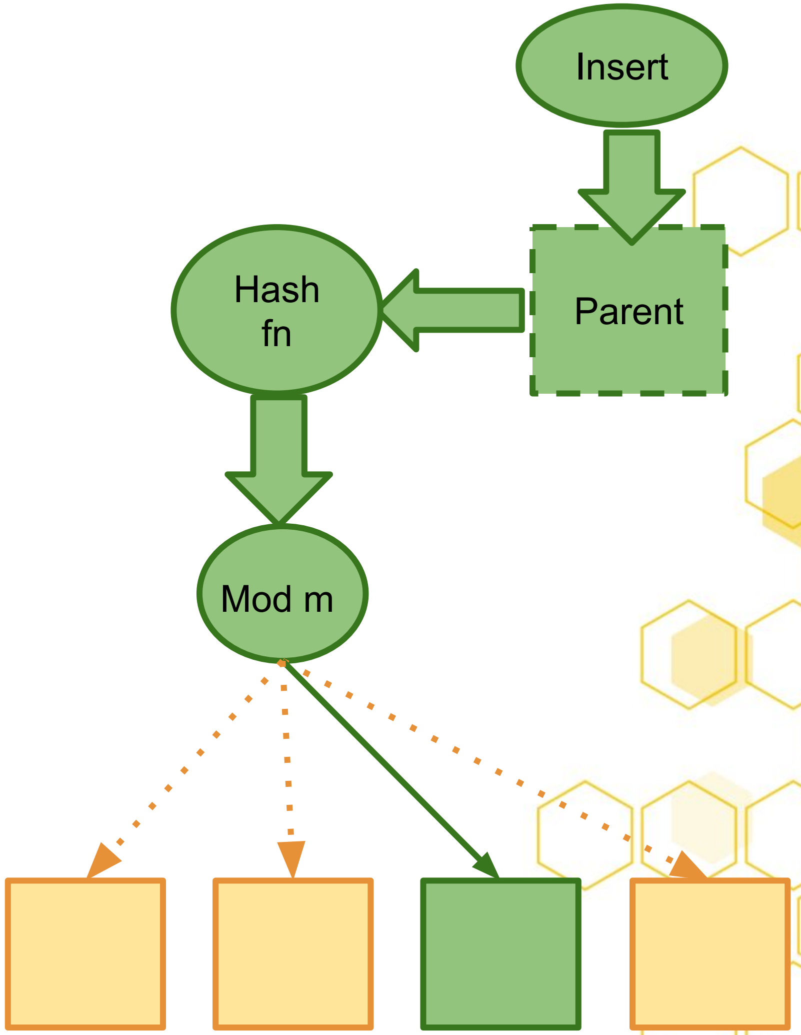
Range Partitioning

- PostgreSQL 10
- Range boundaries
  - Lower inclusive (>=)
  - Upper exclusive (<)
- Unbounded values
  - MINVALUE
  - MAXVALUE



Hash Partitioning

- PostgreSQL 11
- Specify modulus and remainder
  - Modulus - non-zero positive integer
  - Remainder - non-negative integer
  - remainder < modulus
- Rows spread on hash value of the partition key



DECLARATIVE PARTITIONING SYNTAX

- Create partitioned table
- Create partitions
- Add a partition
- Remove a partition

Create Partitioned Table

```
CREATE TABLE parent ( <col list > )
   PARTITION BY <strategy> ( <partition key> );
```

List:

```
CREATE TABLE plist(id int, col1 varchar)
   PARTITION BY LIST (col1);
```

Range:

```
CREATE TABLE prange(id int, col1 int, col2 int)
   PARTITION BY RANGE (col1);
```

Hash:

```
CREATE TABLE phash(id int, col1 int, col2 int)
   PARTITION BY HASH (col1);
```

Create Partitions

```
CREATE TABLE child PARTITION OF parent
    FOR VALUES <partition bounds>
```

List:

```
CREATE TABLE clist PARTITION OF plist
    FOR VALUES IN ('CHENNAI', 'OOTY');
```

Range:

```
CREATE TABLE crange PARTITION OF prange
    FOR VALUES FROM (10) TO (20);
```

Hash:

```
CREATE TABLE chash PARTITION OF phash
    FOR VALUES WITH (MODULUS 5, REMAINDER 0);
```

Add a Partition

```
ALTER TABLE parent ATTACH PARTITION child
FOR VALUES <partition bounds>
```

List:

```
ALTER TABLE plist ATTACH PARTITION clist2
FOR VALUES IN (‘MUMBAI’);
```

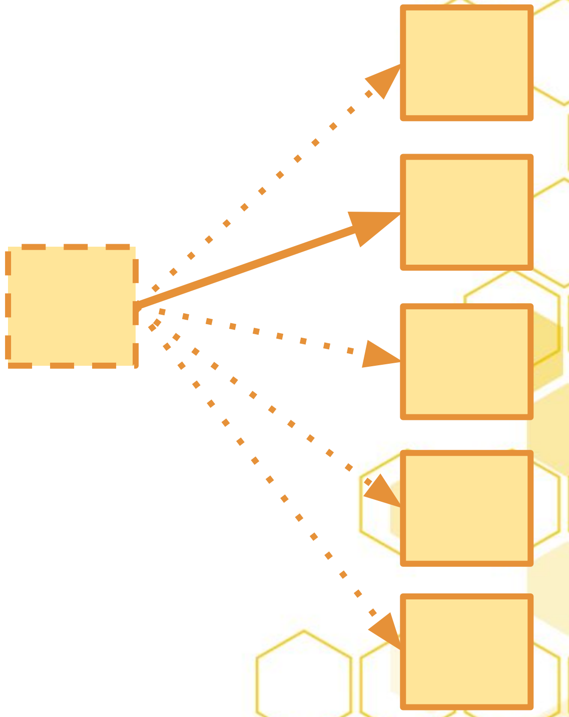
Range:

```
CREATE TABLE prange ATTACH PARTITION
crange2 FOR VALUES FROM (20) TO (50);
```

Hash:

```
CREATE TABLE phash ATTACH PARTITION
chash2 FOR VALUES WITH (MODULUS 5,
REMAINDER 1);
```

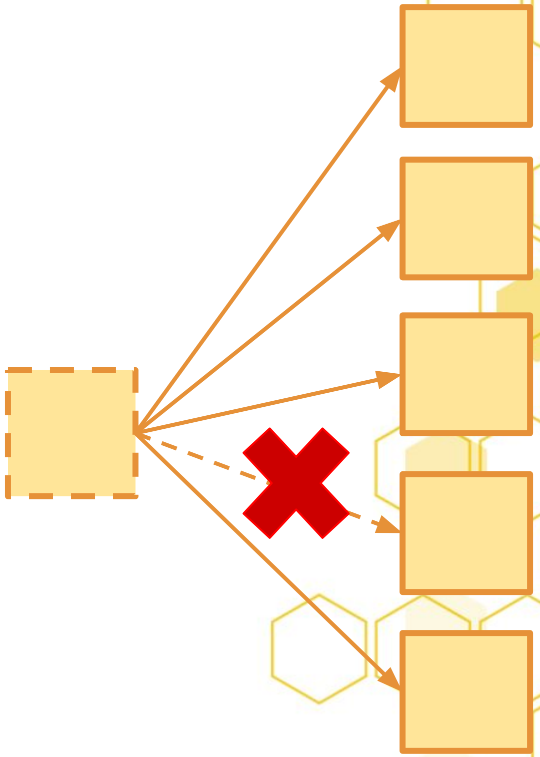
* All entries in the child will be checked to confirm if they meet partition bound.



Remove Partition

```
ALTER TABLE parent
   DETACH PARTITION child;
```

- It will no longer have the partition bound restriction.
- It will retain all the other constraints and triggers.
- If you no longer want the partition data then you may simply use DROP command to remove the table completely.



TYPES OF PARTITIONING

- Multi column partitioning
- Multi level partitioning

Multicolumn Partitioning

- Multiple columns as partition key
- Supported for range and hash
- Column limit : 32

Multicolumn Range Partitioning

- Specify the lower and upper bound for each of the partition key involved.

```
CREATE TABLE prange (col1 int, col2 int, col3 int)
  PARTITION BY RANGE (col1, col2, col3);
```

```
CREATE TABLE crange2 PARTITION OF prange
FOR VALUES FROM (10, 100, 50) TO (500, 500, 150);
```

Multicolumn Range Partitioning

- Every column following `MAXVALUE / MINVALUE` must also be the same.

```
CREATE TABLE crange1 PARTITION OF prange
  FOR VALUES FROM (10, MINVALUE, MINVALUE)
    TO (10, 100, 50);

CREATE TABLE crange3 PARTITION OF prange
  FOR VALUES FROM (500, 500, 150)
    TO (MAXVALUE, MAXVALUE, MAXVALUE);
```

Multicolumn Range Partitioning

- The row comparison operator is used for insert
  - Elements are compared left-to-right, stopping at first unequal pair of elements.
    Consider partition `(0, 0) TO (100, 50)`
    `(0, 199),(100, 49)` fits while
    `(100, 50), (101, 10)` does not.

Multicolumn Hash Partitioning

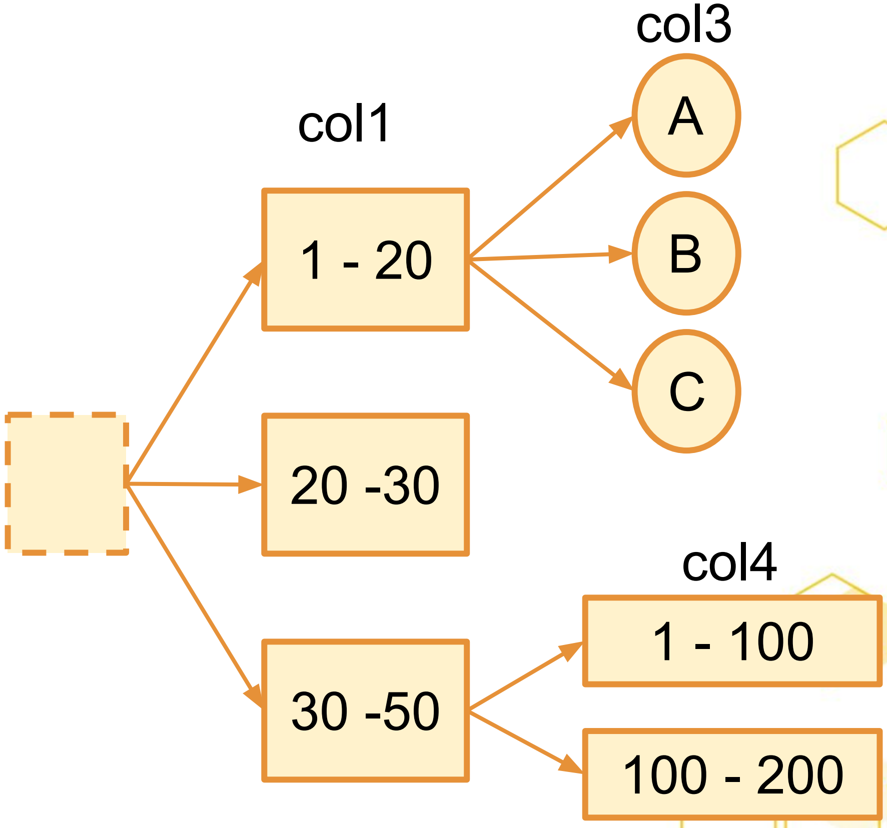
- Only one bound is specified - The hash of each of partition key is calculated and combined to get a single hash value based on which the child partition is determined.

```
CREATE TABLE phash (col1 int, col2 int) PARTITION BY
HASH (col1, col2);

CREATE TABLE phash1 PARTITION OF hparent FOR VALUES
WITH (MODULUS 3, REMAINDER 2);

CREATE TABLE phash2 PARTITION OF hparent FOR VALUES
WITH (MODULUS 3, REMAINDER 1);
```

Multilevel Partitioning

- Different strategies and partition key can be used at different levels.
- Example:

```
CREATE TABLE child1 PARTITION
OF                      parent
FOR VALUES FROM (1) TO (20)
PARTITION BY LIST (col3);
```



BENCHMARKING

- pgbench options
- Bulk load performance
- Read-only query performance
- Sequential scan performance

pgbench options

```
pgbench -i --partitions <integer>
[--partition-method <method>]
```

- `partitions` : positive non-zero integer value
- `partition-method` : Default range. Hash also supported.
- Error if the `--partition-method` is specified without a valid `--partitions` option.
- The pgbench_accounts table is partitioned on `aid`.

pgbench options

- For range partitions, scale is equally split across partitions.
    - lower bound of the first partition is `MINVALUE`,
    - upper bound of the last partition is `MAXVALUE`.

```
pgbench_accounts_1 FOR VALUES FROM (MINVALUE) TO (10001)....
pgbench_accounts_10 FOR VALUES FROM (90001) TO (MAXVALUE)
```

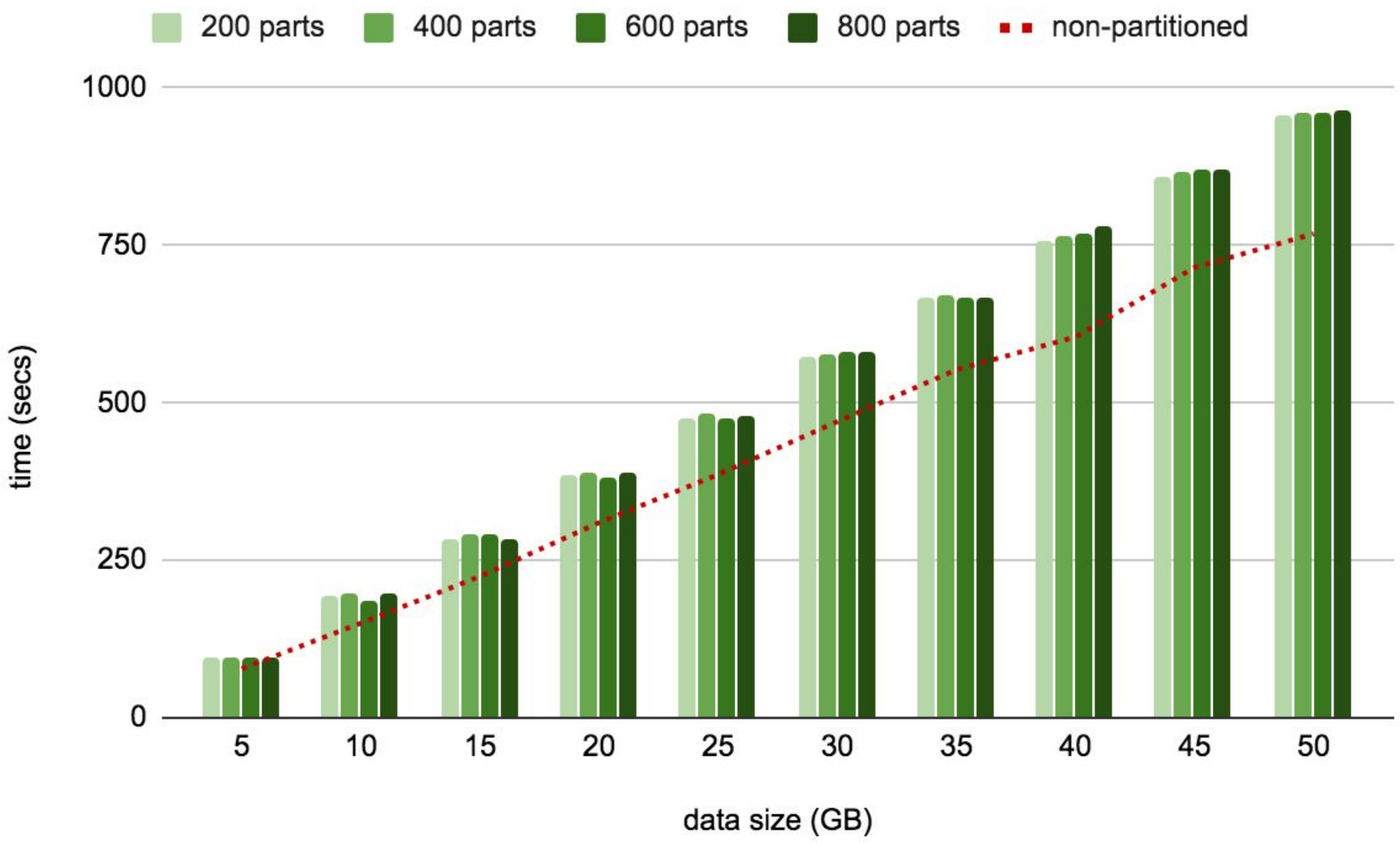
- For hash partitions, the number of partitions specified is used in the modulo operation and the remainder ranges from 0 to partitions - 1

```
pgbench_accounts_1 FOR VALUES WITH (modulus 10, remainder 0)..
pgbench_accounts_10 FOR VALUES WITH (modulus 10, remainder 9)
```

(Examples are using scale 1 and --partitions as 10.)

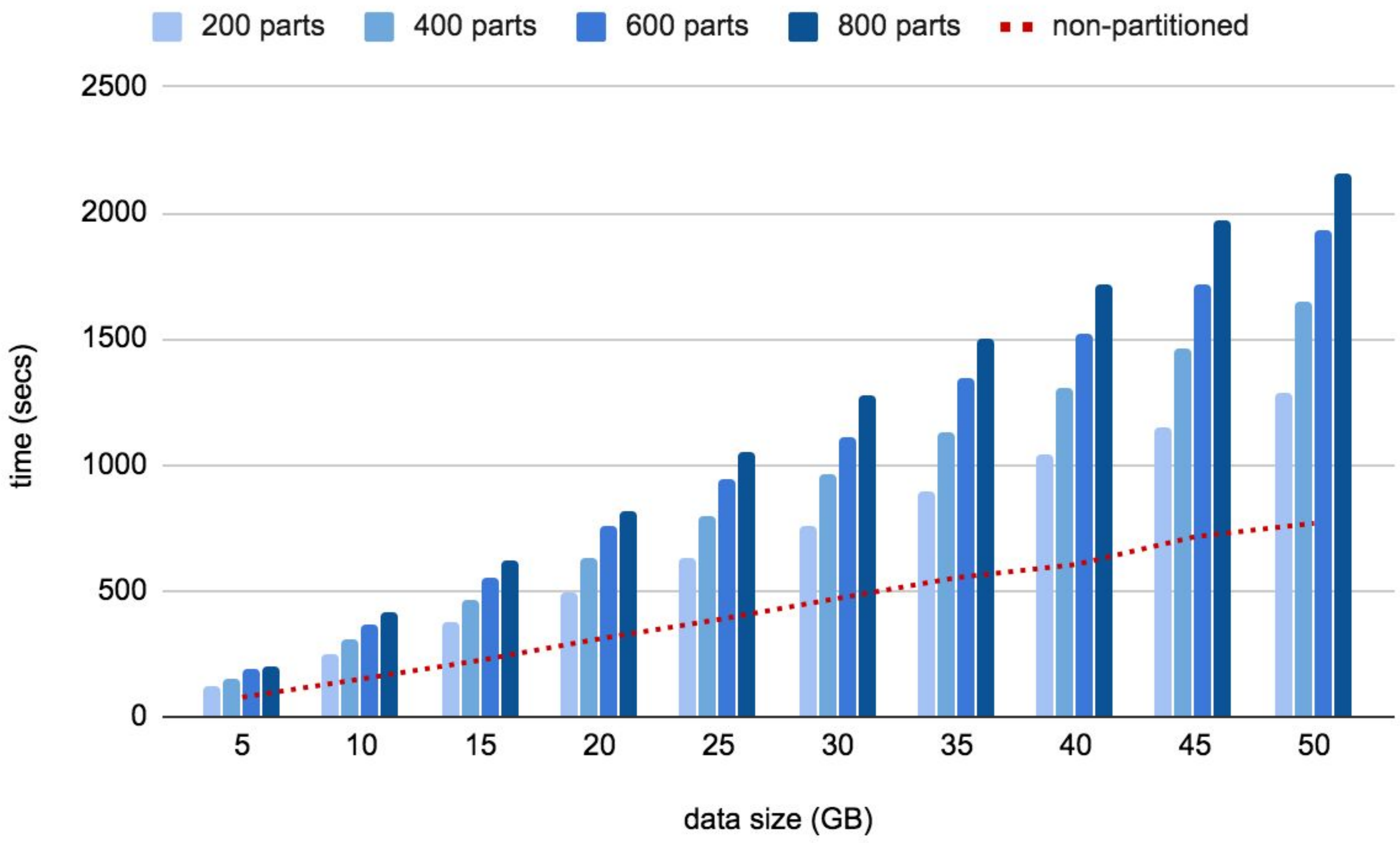
Bulk load: range partitioning

- bulkload command COPY is used to populate accounts table
- range-partitioned table takes a slightly longer time
- partition count hardly influences the load time.



Bulk load: hash partitioning

- number of partitions has heavily impacted the load time
- All partitions (tables) are constantly switched.

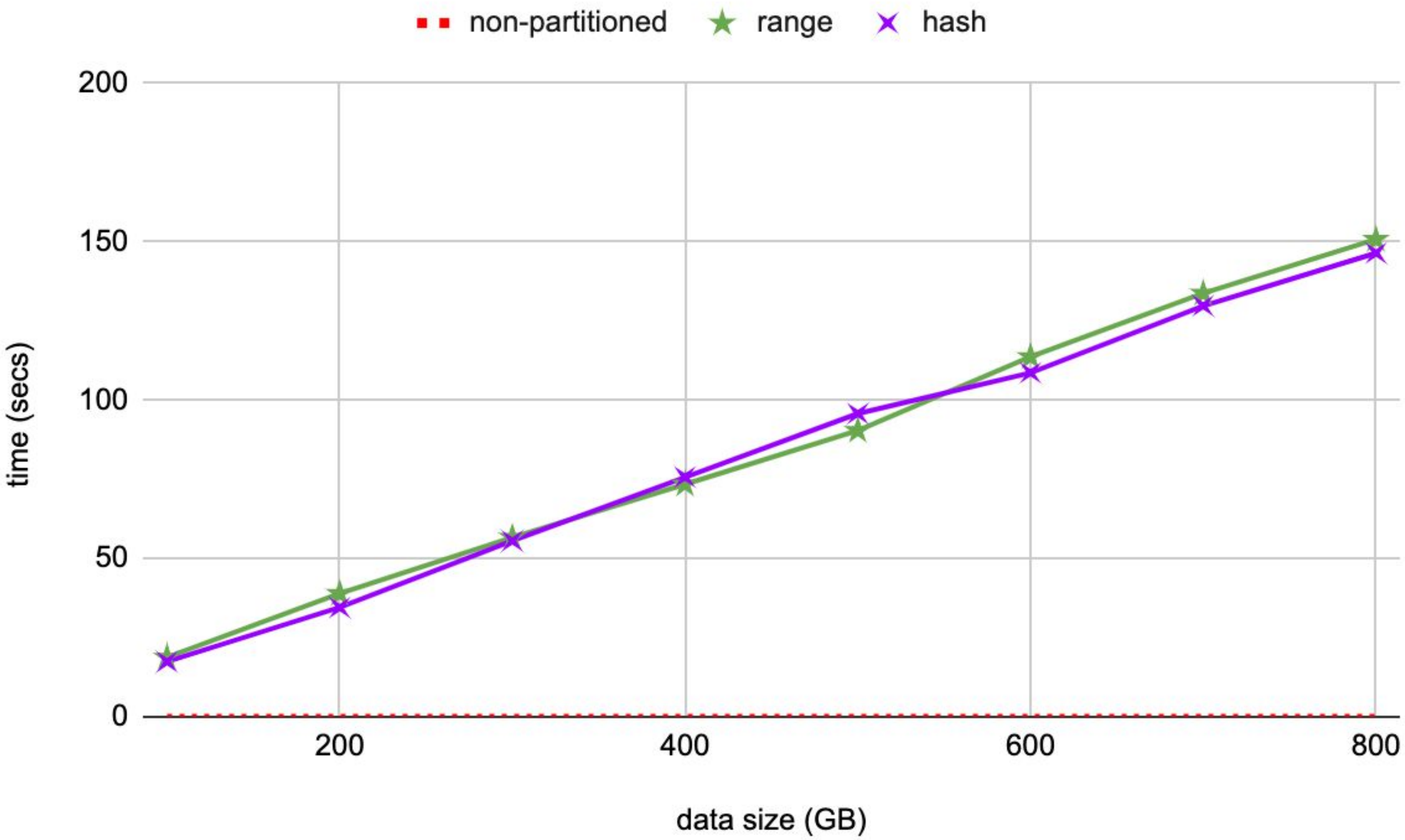


Bulk load: Conclusion

- **data ordered on the partition key column**, no matter the size or the number of partitions, the operation would take about **20–25%** more time than an unpartitioned table.
- If the data being copied is **unordered** with respect to the partition key then the time taken will depend on how often the **partition has to be switched** while insertions.

Seq Scan: default point query

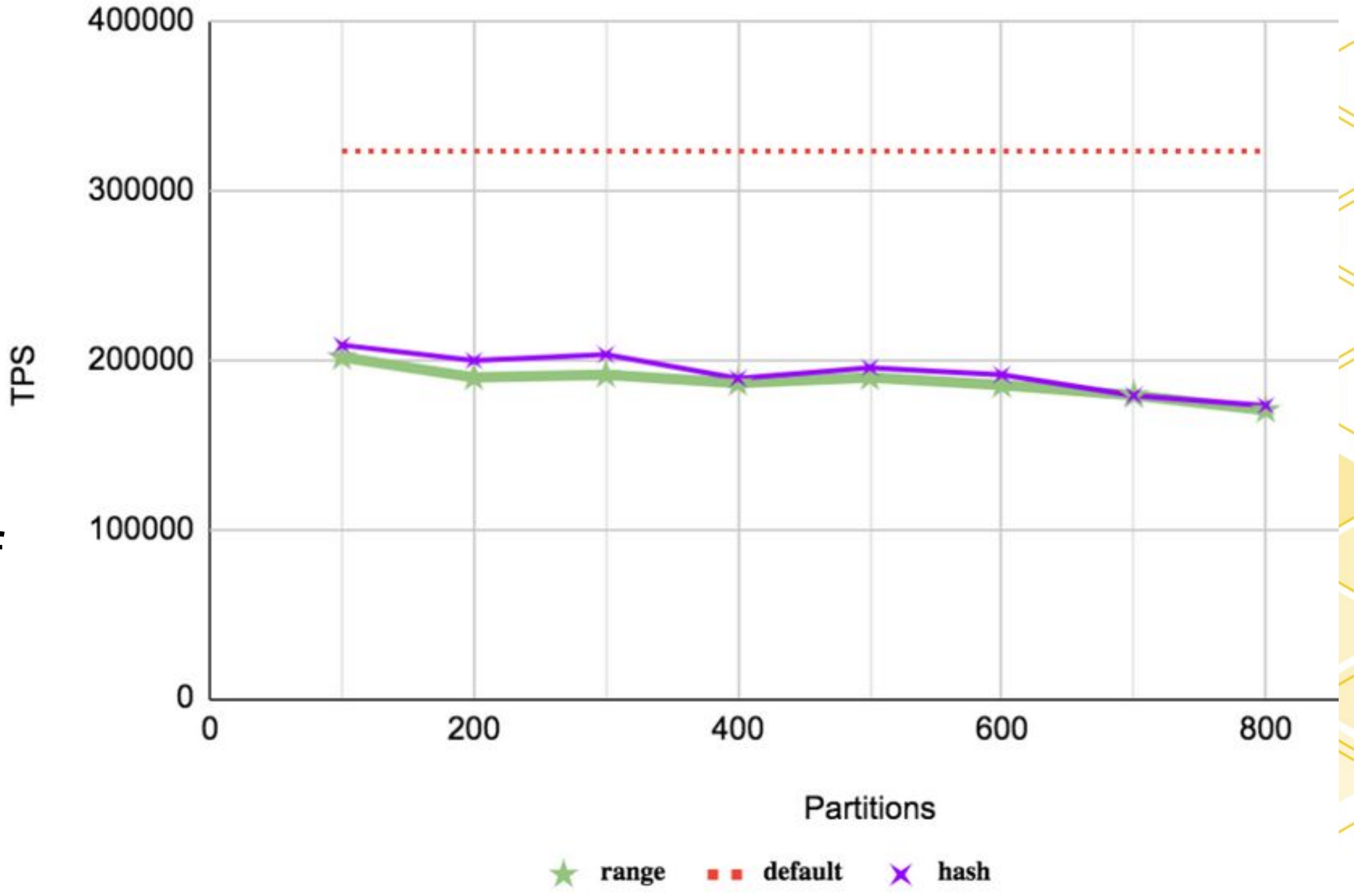
- Remove the index created by pgbench
- non-partitioned table entire data scanned.
- Partition pruning - chosen partition scanned
- ~63 GB data



- The amount of data in each partition reduces as the number of partitions increase

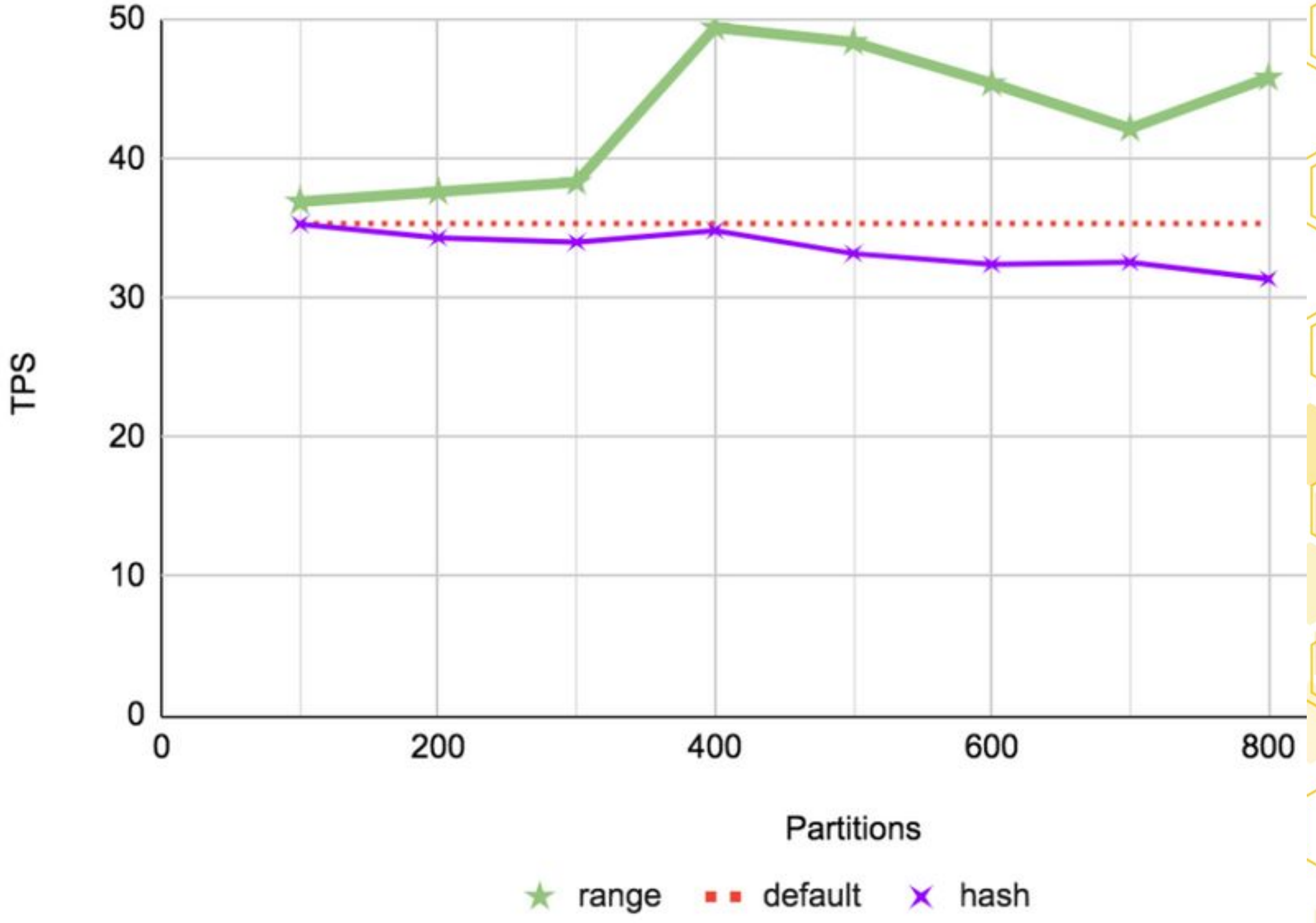
Read-only: default point query

- default query
- ~63GB data + 10GB indexes (scale=5000)
- target only one row in one particular partition.
- 40% drop: overhead of handling of a large number of partitions
- slow degradation as number of partitions increase



Read-only: custom range query

- index scan
- targeting 0.02% rows in sequence
- range: at most two partitions touched
- hash: all partitions touched
- 50 GB



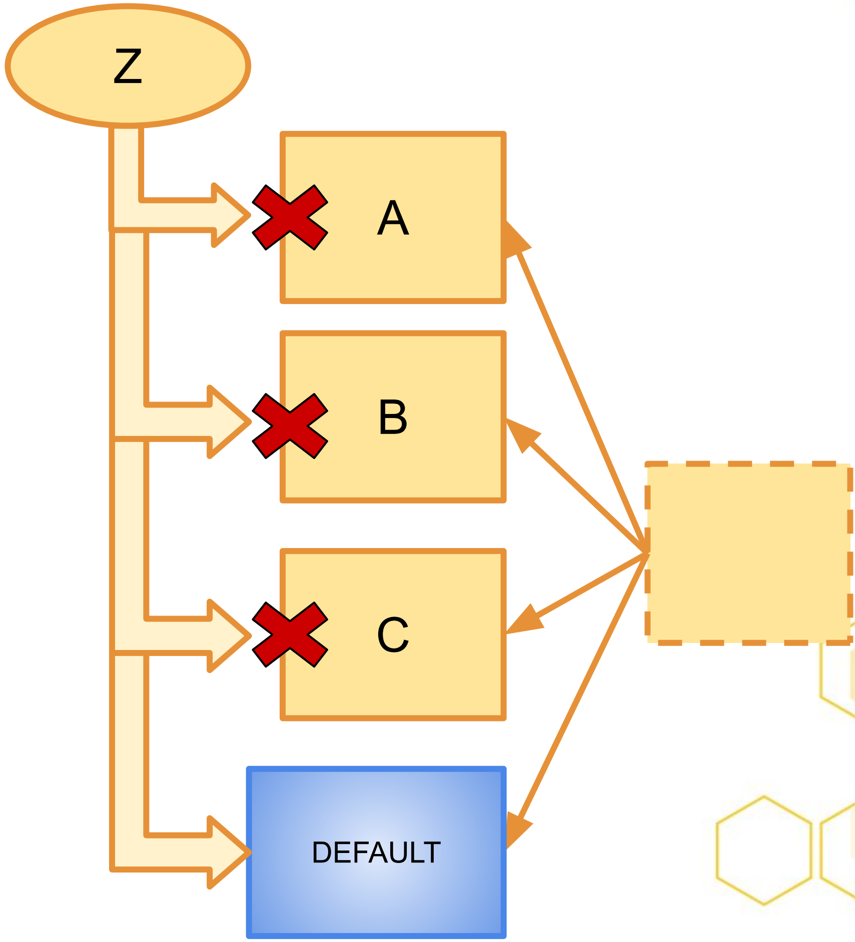
OTHER FEATURES

- Default partitions
- Runtime Partition Pruning
- Partition-wise join
- Partition-level aggregation
- Partition Tree Information

Default Partition

- PostgreSQL 11
- Catch tuples that do not match partition bounds of others.
- Support for: list, range
- Syntax:

```
CREATE TABLE child
    PARTITION OF parent
        DEFAULT;
```



(NOT (col1 IS NOT NULL) AND
(col1 = ANY (ARRAY['A', 'B', 'C'])))

Default Partition

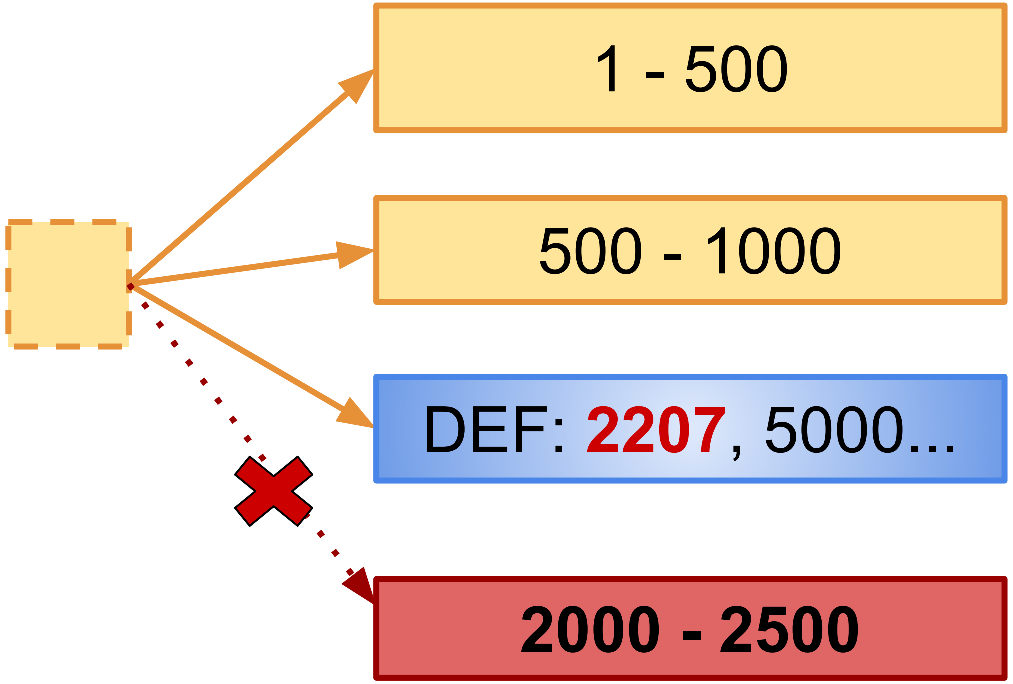
Add new partition

All the rows in default partition are scanned.

Amount of time taken depends on the number of rows in the default partition

Sample:

1382 ms to add partition when one Cr rows in default partition but **2 ms** when it is empty.



```
ERROR:  updated partition constraint for default partition "part_def" would be violated by some row
```

Runtime Partition Pruning

- PostgreSQL 11
- Performed at two levels
    - Executor Initialization - prepared query
    - Actual Execution
- SET enable_partition_pruning. Default is on.

Runtime Partition Pruning

Executor Initialization

```
PREPARE prep (int) as SELECT * from t1 where pkey < $1;
EXPLAIN EXECUTE prep(1500);

 Append  (cost=0.00..168.06 rows=3012 width=8)
   Subplans Removed: 2
   ->  Seq Scan on p1 t1_1  (cost=0.00..38.25 rows=753 width=8)
         Filter: (pkey < $1)
   ->  Seq Scan on p2 t1_2  (cost=0.00..38.25 rows=753 width=8)
         Filter: (pkey < $1)
(6 rows)
```

Runtime Partition Pruning

Actual Execution - Unpartitoined Case

Considering a table with 6000 rows performs nest loop join with another containing 4000 rows but only 2000 rows match the join condition.

```
 Nested Loop (actual rows=1000 loops=1)
   ->  Seq Scan on ltbl (actual rows=6000 loops=1)
   ->  Index Scan using rtbl_pkey on rtbl (actual rows=0 loops=6000)
         Index Cond: (col1 = ltbl.col1)
 Planning Time: 0.248 ms
 Execution Time: 15.265 ms
(6 rows)
```

Runtime Partition Pruning

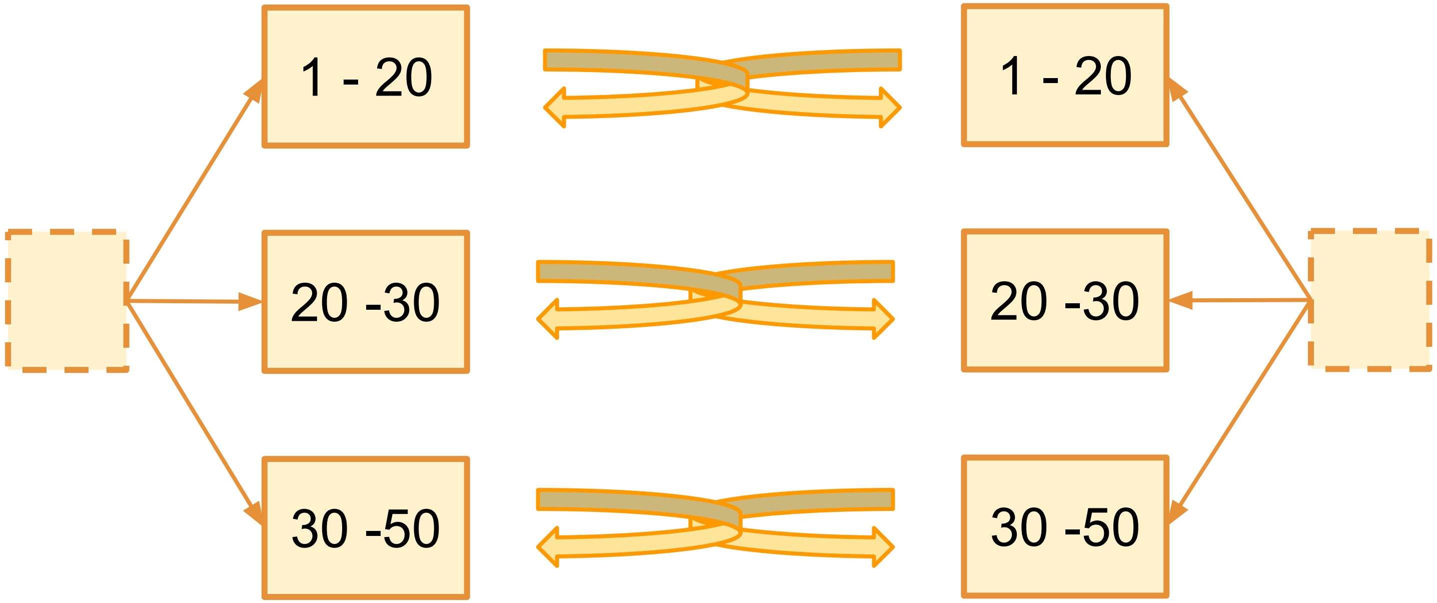
Actual Execution - Partitoined Case

Consider that the table with 4000 rows is partitioned.

```
Nested Loop (actual rows=1000 loops=1)
  ->  Seq Scan on ltbl (actual rows=6000 loops=1)
  ->  Append (actual rows=0 loops=6000)
        ->  Index Scan using p1_pkey on p1 t1_1 (actual rows=1 loops=1500)
              Index Cond: (pkey = ltbl.col1)
        ->  Index Scan using p2_pkey on p2 t1_2 (never executed)
              Index Cond: (pkey = ltbl.col1)
        ->  Index Scan using p3_pkey on p3 t1_3 (actual rows=0 loops=500)
              Index Cond: (pkey = ltbl.col1)
Planning Time: 0.325 ms
Execution Time: 10.632 ms (~30% drop)
```

Partition-wise join

- PostgreSQL 11
- SET enable_partitionwise_join. Default is off.
- Join should be on partition key and both partitions should have same bounds for partitions.



Partition-wise join

```
Hash Join (actual rows=20000 loops=1)
  Hash Cond: (a.col1 = b.col1)
  ->  Append (actual rows=40000 loops=1)
        ->  Seq Scan on a1 a_1 (actual rows=10000 loops=1)
        ->  Seq Scan on a2 a_2 (actual rows=10000 loops=1)
        ->  Seq Scan on a3 a_3 (actual rows=10000 loops=1)
        ->  Seq Scan on a4 a_4 (actual rows=10000 loops=1)
  ->  Hash (actual rows=20000 loops=1)
        Buckets: 32768  Batches: 1  Memory Usage: 1038kB
        ->  Append (actual rows=20000 loops=1)
              ->  Seq Scan on b1 b_1 (actual rows=10000 loops=1)
              ->  Seq Scan on b2 b_2 (actual rows=10000 loops=1)
              ->  Seq Scan on b3 b_3 (actual rows=0 loops=1)
              ->  Seq Scan on b4 b_4 (actual rows=0 loops=1)
Planning Time: 0.119 ms
Execution Time: 37.121 ms
```

Partition-wise join

```
SET enable_partitionwise_join =on;
Append (actual rows=20000 loops=1)
   ->  Hash Join (actual rows=10000 loops=1)
         Hash Cond: (a_1.col1 = b_1.col1)
         ->  Seq Scan on a1 a_1 (actual rows=10000 loops=1)
         ->  Hash (actual rows=10000 loops=1)
               ->  Seq Scan on b1 b_1 (actual rows=10000 loops=1)
   ->  Hash Join (actual rows=10000 loops=1)
         Hash Cond: (a_2.col1 = b_2.col1)
         ->  Seq Scan on a2 a_2 (actual rows=10000 loops=1)
         ->  Hash (actual rows=10000 loops=1)
               ->  Seq Scan on b2 b_2 (actual rows=10000 loops=1)
```

.(repeat for a3, b3 and a4, b4)

.

```
 Planning Time: 0.250 ms
 Execution Time: 19.422 ms
```

Almost 50% reduction in execution time.

Partition-level Aggregation

- PostgreSQL 11
- manually set enable_partitionwise_aggregate (default is off)
- When GROUP BY uses partition key, aggregate individual partition.
- When grouped on non-partition key, PartialAggregate performed on partitions and then combined.
- Aggregate pushed down if partition is foreign table.

Partition-level Aggregation : Example

Table is partitioned on col1 and has 3 partitions with total of 25,000 rows.

```
SELECT col1, count(*) FROM t1 GROUP BY col1;

 HashAggregate (actual rows=8 loops=1)
   Group Key: t1_p1.col1
   ->  Append (actual rows=25000 loops=1)
         ->  Seq Scan on t1_p1 (actual rows=10000 loops=1)
         ->  Seq Scan on t1_p2 (actual rows=6000 loops=1)
         ->  Seq Scan on t1_p3 (actual rows=9000 loops=1)
 Planning Time: 0.193 ms
 Execution Time: 11.498 ms
```

Partition-level Aggregation : Example

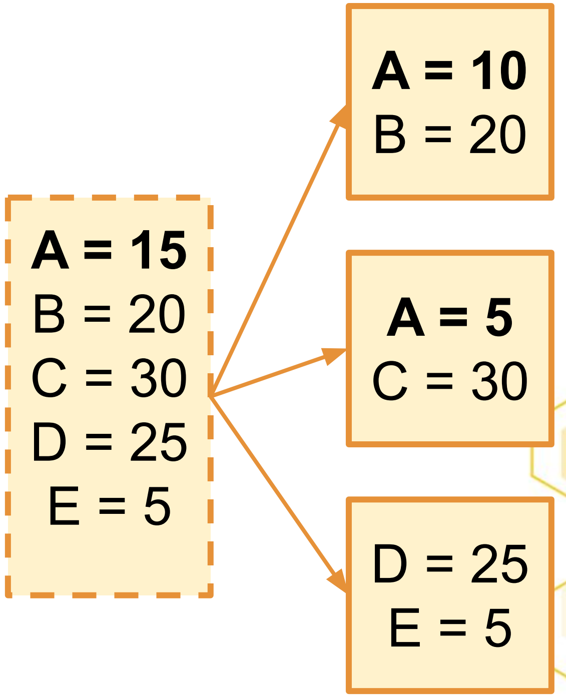
```
SET enable_partitionwise_aggregate=on;

Append (actual rows=7 loops=1)
   ->  HashAggregate (actual rows=2 loops=1)
         Group Key: t1_p1.col1
         ->  Seq Scan on t1_p1 (actual rows=10000 loops=1)
   ->  HashAggregate (actual rows=2 loops=1)
         Group Key: t1_p2.col1
         ->  Seq Scan on t1_p2 (actual rows=6000 loops=1)
   ->  HashAggregate (actual rows=3 loops=1)
         Group Key: t1_p3.col1
         ->  Seq Scan on t1_p3 (actual rows=9000 loops=1)
 Planning Time: 0.161 ms
 Execution Time: 9.046 ms
```

about 20% decrease in execution time

Partition-level Aggregation : Example

When Aggregate does not use partition key

```
Finalize HashAggregate (actual rows=7 loops=1)
  Group Key: t1_p1.col2
  ->  Append (actual rows=10 loops=1)
     ->  Partial HashAggregate (actual rows=2 loops=1)
         Group Key: t1_p1.col2
         ->  Seq Scan on t1_p1 (actual rows=10000 loops=1)
     ->  Partial HashAggregate (actual rows=2 loops=1)
         Group Key: t1_p2.col2
         ->  Seq Scan on t1_p2 (actual rows=6000 loops=1)
     ->  Partial HashAggregate (actual rows=3 loops=1)
         Group Key: t1_p3.col2
         ->  Seq Scan on t1_p3 (actual rows=9000 loops=1)
Planning Time: 0.235 ms
Execution Time: 9.541 ms
```

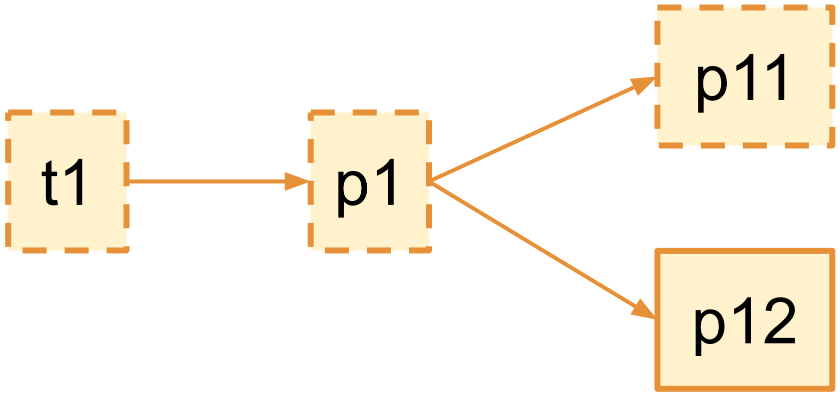


Partition Tree Information

- PostgreSQL 12
- pg_partition_tree: Displays the entire partition tree in table format.
- pg_partition_ancestors: Displays all the ancestors from the partition specified to the root.
- pg_partition_root: Displays the topmost root partitioned table.

Partition Tree Information

```
SELECT * FROM pg_partition_tree ('t1');
 relid | parentrelid | isleaf | level
-------+-------------+--------+-------
 t1    |             | f      |     0
 p1    | t1          | f      |     1
 p11   | p1          | f      |     2
 p12   | p1          | t      |     2
(4 rows)
```



Partition Tree Information

```
SELECT * FROM pg_partition_root('p12');
 pg_partition_root
-------------------
 t1
(1 row)

SELECT * FROM pg_partition_ancestors('p12');
 relid
-------
 p12
 p1
 t1
(3 rows)
```

CONCLUSION

- Partition helps in certain scenarios not all.
- Know your data and experiment to detemine the best partition parameters for your database table.

EDB™ POSTGRES

PGConf India

Thank You

நன்றி

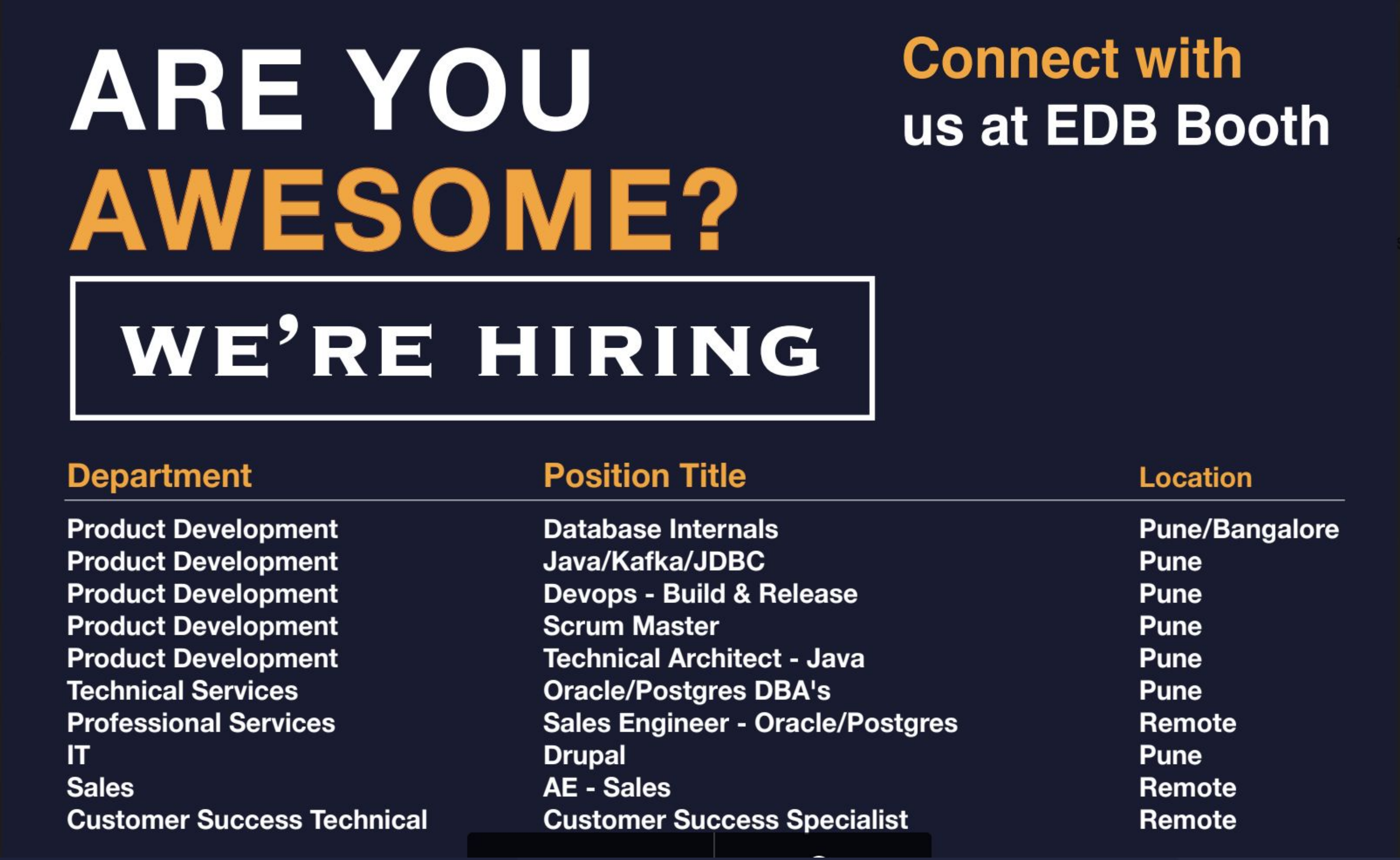
Merci

धन्यवाद

ありがとう

Danke

MBeenaEmerson



ARE YOU AWESOME?

WE'RE HIRING

Connect with us at EDB Booth

Department	Position Title	Location
Product Development	Database Internals	Pune/Bangalore
Product Development	Java/Kafka/JDBC	Pune
Product Development	Devops - Build & Release	Pune
Product Development	Scrum Master	Pune
Product Development	Technical Architect - Java	Pune
Technical Services	Oracle/Postgres DBA's	Pune
Professional Services	Sales Engineer - Oracle/Postgres	Remote
IT	Drupal	Pune
Sales	AE - Sales	Remote
Customer Success Technical	Customer Success Specialist	Remote