Partitioning in PostgreSQL

Why Partitioning

Availability

- Recovering large tables take a long time
- Exclusive locks on maintenance operations

Manageability

- Long index build times
- Purging data
- Securing old data

Performance

- Large tables and indexes do not fit in cache

Table Inheritance

- Before declarative partitioning was available in PostgreSQL 10, table inheritance was used to split large tables
- Table inheritance provided some benefits, but was difficult to maintain
- Performance suffered on data loads

Table Inheritance

- Data is controlled using CHECK contstraints on the child tables

```
CREATE TABLE phone_numbers (
  id           bigserial,
  country_code int,
  area_code    int,
  co           int,
  line         varchar
);

CREATE TABLE phone_numbers_001 (
    CHECK ( country_code <= 1 )
) INHERITS (phone_numbers);
```

Table Inheritance

- Data is routed to the child tables using triggers

```
CREATE OR REPLACE FUNCTION phone_numbers_insert_trigger()
  RETURNS TRIGGER AS
$$
BEGIN
    IF ( NEW.country_code <= 1) THEN
        INSERT INTO phone_numbers_001 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Country code out of range.';
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Availability Benefits of Partitioning

- Backup and restore commands like pg_dump and pg_restore can be run in individual partitions

- Maintenance operations like `VACUUM FULL` which take Access Exclusive Locks can be performed on individual partitions

Manageability Benefits of Partitioning

- Rolling windows of data can be created by adding new partitions and dropping old ones
  - Does not create bloat with deleting expired rows
- Build smaller indexes on individual partitions
  - Can have different indexes on different partitions
- Security privileges can be applied at the partition level

Performance Benefits of Partitioning

- Whole partitions can be pruned away at query time so a smaller subset of data is scanned

- Joins can be optimized by joining by partition

Partition Pruning

- If the query is constructed properly, PostgreSQL can eliminate whole partitions from being scanned

```
SELECT sum(o_totalprice)
  FROM orders
 WHERE o_orderdate
   BETWEEN '1992-04-01'
       AND '1992-06-30';
```

Partitioning Strategies

Range Partitioning

- Data is placed in partitions based on a range of values

List Partitioning

- Data is placed in partitions based on a list of discrete values

Hash Partitioning

- Data is placed in partitions based on a hash algorithm applied to a key

IMPLEMENTING PARTITIONED TABLES

Partition Type

The Partition type is declared in the PARTITION clause

```
CREATE [ { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [
... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression
) }
[ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
```

Partition Value

Values contained by the partition are defined in the partition definition

```
CREATE [ { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name
    PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] { FOR VALUES partition_bound_spec | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression
) }
[ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
```

Partition Key

- The key must be made up of columns and/or immutable expression
- The key value must be a literal
- The partition key can include up to 32 columns or expressions
  - List partitioning is limited to a single column

Range Partitioning

- Can not have overlapping ranges
- The special values MINVALUE and MAXVALUE can be used to indicate that there is no lower or upper bound
- The value can not be NULL

Range Partitioning

```
CREATE TABLE phone_numbers (
  id           bigserial,
  country_code int,
  area_code    int,
  co           int,
  line         varchar
)
PARTITION BY RANGE (country_code);

CREATE TABLE phone_numbers_001
  PARTITION OF phone_numbers
    FOR VALUES FROM (1) TO (19);
```

Unbounded Ranges

Use MINVALUE and MAXVALUE to indicate no lower or upper bounds

```
CREATE TABLE phone_numbers_min
  PARTITION OF phone_numbers
    FOR VALUES FROM (MINVALUE) TO (1);

CREATE TABLE phone_numbers_max
  PARTITION OF phone_numbers
    FOR VALUES FROM (19) TO (MAXVALUE);
```

Multicolumn Range Partitioning

- Up to 32 columns can be used for a composite key

- Order is significant

```
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

```
CREATE TABLE phone_numbers (
  id           bigserial,
  country_code int,
  area_code    int,
  co           int,
  line         varchar
)
PARTITION BY RANGE (country_code, area_code);

CREATE TABLE phone_numbers_001_0
  PARTITION OF phone_numbers
    FOR VALUES FROM (1, MINVALUE) TO (1, 500);

CREATE TABLE phone_numbers_001_500
  PARTITION OF phone_numbers
    FOR VALUES FROM (1, 500) TO (1, MAXVALUE);
```

List Partitioning

- Only a single column can be used for a key
- Not all partitions need to be defined
- Allows NULL values

List Partitioning

```
CREATE TABLE phone_numbers (
  id           bigserial,
  country_code int,
  area_code    int,
  co           int,
  line         varchar
)
PARTITION BY LIST (country_code);

CREATE TABLE phone_numbers_001
  PARTITION OF phone_numbers
    FOR VALUES IN (1);

CREATE TABLE phone_numbers_002
  PARTITION OF phone_numbers
    FOR VALUES IN (2, 3, 4, 5, 6, 7, 8, 9, 10);
```

NULL Values

NULL values for the partition key are allowed with list partitioning

```
CREATE TABLE phone_numbers_null
  PARTITION OF phone_numbers
    FOR VALUES IN (null);
```

Hash Partitioning

- Not all partitions need to be defined
- The modulous for each partition should be the same but not required
  - Allows for incrementally increasing the number of partitions
- Allows NULL values
  - They are placed in the first partition

Hash Partitioning

```
CREATE TABLE users (
  id          bigserial,
  name        varchar,
  email       varchar,
  created_at timestamptz DEFAULT CURRENT_TIMESTAMP
)
PARTITION BY HASH (email);

CREATE TABLE users_02_00
  PARTITION OF users
    FOR VALUES WITH (MODULUS 2, REMAINDER 0);

CREATE TABLE users_02_01
  PARTITION OF users
    FOR VALUES WITH (MODULUS 2, REMAINDER 1);
```

Sub Partitioning

- Not all partitions need to be sub partitioned the same way
- The sub partitions can be Range, List or Hash
- There is no limit to how many levels a table can be sub partitioned

Sub Partitioning

```
CREATE TABLE phone_numbers (
  id           bigserial,
  country_code int,
  area_code    int,
  co           int,
  line         varchar
)
PARTITION BY LIST (country_code);

CREATE TABLE phone_numbers_001
  PARTITION OF phone_numbers
    FOR VALUES IN (1)
  PARTITION BY HASH (area_code);

CREATE TABLE phone_numbers_001_0
  PARTITION OF phone_numbers_001
    FOR VALUES WITH (MODULUS 2, REMAINDER 0);
```

Default Partitions

- Used for values not fitting any other existing partition
- Available for Range and List partitioning types
- There can only be one per parent table

```
CREATE TABLE phone_numbers_default
  PARTITION OF phone_numbers
  DEFAULT;
```

Verifying Partitioning

- Examine the hidden column `tableoid` to see the source partition for each row

```
=> SELECT tableoid, line FROM phone_numbers LIMIT 3;
 tableoid |        line
----------+--------------------
   519539 | 1 (202) 456-1111
   519546 | 1 (888) 280-4331
   519618 | 44-020-7930-4832
(3 rows)
```

Limitations

- Primary keys must contain the partition key
- It is not possible to create an exclusion constraint spanning all partitions
- Foreign keys referencing partitioned tables are not supported
- BEFORE ROW triggers must be defined on individual partitions, not the partitioned table
- When an UPDATE causes a row to move from one partition to another, there is a chance that another concurrent UPDATE or DELETE will get a serialization failure error

MAINTENANCE OF PARTITIONED TABLES

Overview

- Maintenance commands like `ALTER TABLE` and `CREATE INDEX` can be performed at the table level or the partition level
- Partitions can be added and removed
- Partitions can be split and merged

Modifying a Table

- A partitioned table can be renamed like an unpartitioned table
  - The partitions are not renamed

- Renaming a column is propagated to all partitions

```
ALTER TABLE phone_numbers RENAME TO phone_numbers2;

ALTER TABLE phone_numbers RENAME COLUMN co TO central_office;
```

Creating Indexes

- There is no concept of a global index

- All indexes created on the parent table are created locally on the partitions

```
CREATE INDEX country_area_code_idx
  ON phone_numbers (country_code, area_code);
```

Creating Indexes

- Indexes created on partitions are local to the partition

```
CREATE INDEX default_co_idx
  ON phone_numbers_default (co);
```

Modifying Columns

- Adding or removing columns propagate to all partitions

- Individual partitions can not have different columns from the parent

```
ALTER TABLE phone_numbers ADD COLUMN phone_type varchar;
```

Removing Partitions

- Individual partitions can be simply dropped

- This takes an Access Exclusive lock on the parent

```
DROP TABLE users_02;
```

Removing Partitions

- Partitions can be DETACHED
- The data still exists but is no longer part of the parent table
- This takes an Access Exclusive lock on the parent

```
ALTER TABLE users DETACH PARTITION users_02;
```

Attaching Partitions

- Partitions can be ATTACHED already containing data
- Existing data must conform to the partition key constraint
- Any missing indexes will be created
- This takes an Access Exclusive lock on the parent

```
CREATE TABLE phone_numbers_044
  (LIKE phone_numbers INCLUDING DEFAULTS INCLUDING CONSTRAINTS);

ALTER TABLE phone_numbers ATTACH PARTITION phone_numbers_044
    FOR VALUES IN (44);
```

Splitting Partitions

- Splitting a partition creates two or more new partitions with data from the original partition
- No single command to split a partition
- Requires some locking of the partition being split for writes

Splitting Partitions

- The new partitions need to be created as stand alone tables

```
CREATE TABLE users_04_00
  (LIKE users_02_00 INCLUDING DEFAULTS INCLUDING CONSTRAINTS);

CREATE TABLE users_04_02
  (LIKE users_02_00 INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
```

Splitting Partitions

- Load the data into the new partitions
- Remove the original partition
- Attach the new partitions

```
BEGIN;
LOCK TABLE users_02_00 IN EXCLUSIVE MODE;
INSERT INTO users_04_00
  SELECT * FROM users_02_00
   WHERE satisfies_hash_partition('users'::regclass, 4, 0, email);
INSERT INTO users_04_02
  SELECT * FROM users_02_00
   WHERE satisfies_hash_partition('users'::regclass, 4, 2, email);
DROP TABLE users_02_00;
ALTER TABLE users ATTACH PARTITION users_04_00 FOR VALUES WITH (MODULUS
4, REMAINDER 0);
ALTER TABLE users ATTACH PARTITION users_04_02 FOR VALUES WITH (MODULUS
4, REMAINDER 2);
COMMIT;
```

Merging Partitions

- Merging partitions creates a single partition from two or more partitions
- No single command to merge a partition
- Requires locking of the parent table

Merging Partitions

- The larger of the 2 partitions should remain
- The data from the smaller partition is moved into the larger partition

```
BEGIN;
ALTER TABLE phone_numbers DETACH PARTITION phone_numbers_002;
ALTER TABLE phone_numbers DETACH PARTITION phone_numbers_011;
INSERT INTO phone_numbers_002 SELECT * FROM phone_numbers_011;
ALTER TABLE phone_numbers
  ATTACH PARTITION phone_numbers_002
  FOR VALUES IN (2, 3, 4, 5, 6, 7, 8, 9, 10, 11);
DROP TABLE phone_numbers_011;
COMMIT;
```

PARTITIONING USAGE

Using Partitioned Tables

- Applications do not need to be aware that the table is partitioned
  - They are referenced like an ordinary table
- Partitioning pruning is automatic
- Partition-wise joins are automatic

Pruning

- Pruning can occur when the key is referenced directly in the WHERE clause or via a join
- Range and List
  - Ranges, equalities and IN lists
- Hash
  - Equalities and IN lists

Pruning

When the key is referenced directly in the WHERE clause, the pruning takes place at planning time

```
=> EXPLAIN SELECT * FROM users WHERE email IN ('jeff@amazon.com');
                                        QUERY PLAN
-------------------------------------------------------------------------
-----------------
 Append  (cost=0.27..8.30 rows=1 width=52)
   ->  Index Scan using users_02_01_email_idx on users_02_01
         (cost=0.27..8.29 rows=1 width=52)
         Index Cond: ((email)::text = 'jeff@amazon.com'::text)
```

Pruning

Pruning takes place at execution time for joins

```
=> EXPLAIN ANALYZE
SELECT users.*
  FROM users
 INNER JOIN employees ON (users.email = employees.email)
 WHERE employees.title = 'CEO';
                                     QUERY PLAN
-------------------------------------------------------------------------------------
Nested Loop (cost=0.27..95.17 rows=4 width=52) (actual time=0.042..0.042 rows=1 loop...
->  Seq Scan on employees  (cost=0.00..20.62 rows=4 width=32) (actual time=0.004..0....
        Filter: ((title)::text = 'CEO'::text)
  ->  Append  (cost=0.27..18.61 rows=3 width=52) (actual time=0.036..0.036 rows=0 lo...
        ->  Index Scan using users_02_01_email_idx on users_02_01  (cost=0.27..6.29 ...
              Index Cond: ((email)::text = (employees.email)::text)
        ->  Seq Scan on users_04_00  (cost=0.00..6.12 rows=1 width=53) (never executed)
              Filter: ((employees.email)::text = (email)::text)
        ->  Seq Scan on users_04_02  (cost=0.00..6.18 rows=1 width=52) (never executed)
              Filter: ((employees.email)::text = (email)::text)
```

Partition-wise Operations

- Joins and aggregates can occur at the partition level
- Speeds up queries by minimizing the amount of data pushed up the stages of the execution tree
- Uses more CPU and memory during planning so disabled by default

Partition-wise Joins

- The partition conditions of the joined tables must match exactly

```
=> EXPLAIN SELECT p_name, ps_suppkey FROM part INNER JOIN partsupp
     ON (p_partkey = ps_partkey) WHERE p_partkey IN (67, 98, 133) ORDER BY 1, 2;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Sort  (cost=262.07..262.13 rows=24 width=37)
   Sort Key: part_p5.p_name, partsupp_p5.ps_suppkey
   ->  Append  (cost=0.86..261.52 rows=24 width=37)
         ->  Nested Loop  (cost=0.86..128.04 rows=12 width=37)
               ->  Index Scan using part_p5_pkey on part_p5  (cost=0.43..17.34 rows...
                     Index Cond: (p_partkey = ANY ('{67,98,133}'::integer[]))
               ->  Index Only Scan using partsupp_p5_pkey on partsupp_p5  (cost=0.4...
                     Index Cond: (ps_partkey = part_p5.p_partkey)
         ->  Nested Loop  (cost=0.86..133.37 rows=12 width=37)
               ->  Index Scan using part_p6_pkey on part_p6  (cost=0.43..17.34 rows...
                     Index Cond: (p_partkey = ANY ('{67,98,133}'::integer[]))
               ->  Index Only Scan using partsupp_p6_pkey on partsupp_p6  (cost=0.4...
                     Index Cond: (ps_partkey = part_p6.p_partkey)
```

Partition-wise Aggregations

- Full aggregation can performed on each partition if the partition key is in the GROUP BY clause

```
=> EXPLAIN SELECT c_nationkey, sum(c_acctbal::numeric)
->    FROM customers
->   WHERE c_nationkey IN (8, 9, 10) GROUP BY c_nationkey;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=116420.28..237196.61 rows=10 width=36)
   ->  HashAggregate  (cost=116420.28..116420.34 rows=5 width=36)
         Group Key: customers_4.c_nationkey
         ->  Seq Scan on customers_4  (cost=0.00..111856.41 rows=608516 width=8)
               Filter: (c_nationkey = ANY ('{8,9,10}'::integer[]))
   ->  HashAggregate  (cost=120776.16..120776.22 rows=5 width=36)
         Group Key: customers_2.c_nationkey
         ->  Seq Scan on customers_2  (cost=0.00..111805.95 rows=1196028 width=8)
               Filter: (c_nationkey = ANY ('{8,9,10}'::integer[]))
```

Summary

- Partitioning tables adds additional maintenance burden
- Implementing partitioning is transparent to clients
- Knowledge of the partition key can produce better performing queries