VISCOSITY

PROFESSIONAL SERVICES

Unlock the Power of PostgreSQL Table Partitioning

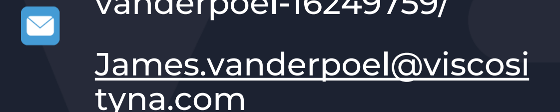
New York Oracle Users Group + Viscosity Webinar

July 2024

James Vanderpoel

Senior Principal Consultant

Viscosity North America



James Vanderpoel

Senior Principal Consultant

@viscosityna

linkedin.com/in/james-vanderpoel-16249759/

James.vanderpoel@viscosityna.com

James Vanderpoel has served in the Senior Principal Consultant Role for Viscosity North America since August of 2023.

- He specializes in PostgreSQL as well as SQL Server, Azure Cloud, and all things Data Related in the Microsoft Ecosystem.
- Prior to his role at Viscosity, he served as a Senior Database Administrator in DevOps/ Production for a number of companies in industries ranging from Finance to Energy, going all the way back to 2016.

Agenda

01: What Is Table Partitioning and Why Use Such

02: Types of Partitioning in PostgreSQL

03: Partitioning Using Inheritance

04: Declarative Partitioning

05: Key Concepts, Best Practices

What is Table Partitioning And Why Use Such?

Partitioning is a technique that allows us to split large tables into smaller tables in a way that is transparent to the client program.

What is Table Partitioning and Why Use Such?

Table partitioning is a technique that improves query performance and data management for large datasets.

Key Benefits:

- **Improved Query Performance**: Partitioning enables faster query execution times through partition pruning, reducing the amount of data to be scanned.
- **Easier Data Management**: Partitioning simplifies the management of large datasets by splitting them into manageable partitions, streamlining actions like archiving, purging, and backup/restore operations.

Main Use Cases:

- Large datasets with varying query patterns
- Data archiving and purging requirements
- Performance optimization for specific queries or workloads

What is Table Partitioning and Why Use Such?

- **Enhanced Data Loading and Indexing**: Partitioning enables parallel data loading and more efficient indexing, leading to faster data ingestion and improved query performance.
- **Cost-Effective Storage**: Partitioning allows for storing less frequently accessed data on cheaper storage media, while keeping frequently accessed data on faster devices.
- **Additional Performance Gains**: Smaller partitioned tables and indexes lead to:
  - Higher cache hit rates and reduced IO
  - Faster vacuum processes with minimized execution time
  - Reduced disk space usage during vacuum full operations
- **Optimize your data management and performance with Postgres Partitioning!**

Types of Table Partitioning in PostgreSQL

Partitioning using Inheritance

Declarative Partitioning

Types of Table Partitioning in PostgreSQL

Declarative Partitioning

Introduced in PostgreSQL 10, Declarative Partitioning is the recommended method for partitioning tables. It supports three partitioning types:

- Range- Partitioned into "Ranges" defined by a Key Column, with lower and upper boundaries
- List- Partitioned by explicitly listing which key values appear in each partition
- Hash- 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

Legacy Partitioning using Table Inheritance

Available prior to PostgreSQL 10

- This method uses:
  - Constraints to define partitions
  - Rules or triggers to route data to the appropriate partition
- Key Characteristics:
  - Child tables can have additional columns not present in the parent table
  - Offers flexible partitioning beyond Range, List, and Hash options

Note: This method is no longer the recommended partitioning approach, superseded by Declarative Partitioning in PostgreSQL10.

Partitioning using Inheritance

viscosityna.com

@ViscosityNA

Partitioning Using Table Inheritance

What's the Object-Oriented Way of Getting Rich?......Inheritance!

PostgreSQL applies object-oriented inheritance to database tables. Here's how:

- Define a parent table (Table_1) and child tables (Table_2)
- The child table inherits from the parent table
- All records in the child table (Table_2) are automatically accessible through the parent table (Table_1)
- See below!

Create Parent and Child Tables:

```
CREATE TABLE Employees_Parent (
Emp_ID INT NOT NULL,
EmpName varchar(25));

CREATE TABLE EmpChild_1_100() INHERITS (Employees_Parent);

CREATE TABLE EmpChild_101_200() INHERITS (Employees_Parent);
```

Partitioning Using Table Inheritance

- Add **Non-Overlapping** Table Constraints to the Child Tables to Define Allowed Key Values in Each:

```
ALTER TABLE EmpChild_1_100 ADD CONSTRAINT Chk_ID CHECK(EmpID BETWEEN 1 AND 100)
ALTER TABLE EmpChild_101_200 ADD CONSTRAINT Chk_ID CHECK(EmpID BETWEEN 101 AND 200)
```

- Create Function To Be Called By Trigger & Trigger

Function:

```
CREATE FUNCTION fn_insert_Employee_trigger()
RETURNS Trigger AS $$
BEGIN
IF NEW.EmpID BETWEEN 1 AND 100
THEN INSERT INTO EmpChild_1_100 (EmpID, EmpName)
VALUES (NEW.EmpID, NEW.EmpName);
ELSIF NEW.EmpID BETWEEN 101 AND 200
THEN INSERT INTO EmpChild_101_200 (EmpID, EmpName)
VALUES (NEW.EmpID, NEW.EmpName);
END IF;
RETURN NULL;
END;
$$
LANGUAGE 'plpgsql';
```

Trigger:

```
CREATE Trigger trg_insert_Employees_Prnt
BEFORE INSERT ON Employees_Parent
FOR EACH ROW EXECUTE FUNCTION
fn_insert_Employee_trigger();
```

Then Insert Values into Parent Table:

```
INSERT INTO Employees_Parent (EmpID, EmpName)
VALUES (1,'James'), (2,'Ali'),(3,'Miles’);
```

Take Note of Where Data Is

```
SELECT * FROM Employees_Parent;
```

ages Notifications Data Output

empid integer	empname character varying (25)
1	James
2	Ali
103	Miles

```
SELECT * FROM EmpChild_1_100;
```

ages Notifications Data Output

empid integer	empname character varying (25)
1	James
2	Ali

```
SELECT * FROM EmpChild_101_200;
```

ages Notifications Data Output

empid integer	empname character varying (25)
103	Miles

What is the difference between and Introverted Engineer and an Extroverted Engineer?
The Extroverted Engineer stares at your shoes when he talks to you!

Declarative Partitioning

Enter Declarative Partitioning

Introduced in Version 10 and enhanced in subsequent versions, Declarative Partitioning offers improved ease of use, performance, and features. This is the preferred method.

- **Partitioned Table**: The table to be divided, which is virtual and storage-less.
- **Partitioning Method**: Choose from hash, range, or list partitioning.
- **Partition Key**: Specify column(s) or expressions to determine partitioning.

How it Works:

- The partitioned table is considered a “Virtual” table, having no storage of its own. Underlying partitions (regular tables associated with partitioned tables) store the data.
- Rows are routed to the correct partition based on the partition key.
- Updating the partition key may migrate a row to a different partition if it no longer meets the original partition's boundaries.

Declarative Partitioning

Advanced Partitioning Features

- **Sub-Partitioning for Enhanced Partition-Pruning**
  - Partitions can themselves be defined as partitioned tables, enabling sub-partitioning and potential additional gains in specific use cases.
- **Partition Flexibility**
  - Partitions can have unique indexes, constraints, and default values.
  - Partitions can be foreign tables but require careful management to ensure partition rule compliance.
- **Partitioned Table Management**
  - Partitioned Tables are virtual and cannot be converted to/from regular tables.
  - Use ATTACH PARTITION and DETACH PARTITION to add/remove partitions, converting them to standalone tables.
- **Note:** Table constraints for partition boundaries are implicitly created but can be manually defined in conjunction with partition management actions.

Declarative Partitioning DDL

1st:

CREATE TABLE <<table_name>>

(<<column>>,

<<column>>)

Partition BY <<Pttn_Method>>(<<partition_key_column(s)>>);

The partition method can again be RANGE, LIST, and HASH.

2nd:

DDL for Range Partitions:

CREATE TABLE <<table_name>>
PARTITION of <<Partitioned Table>>
FOR VALUES FROM <<lower_bound>>
TO <<upper_bound>>

DDL for List Partitions:

CREATE TABLE <<table_name>>
PARTITION of <<Partitioned Table>>
FOR VALUES IN <<Partition_Value>>

DDL for HASH Partitions:

CREATE TABLE <<table_name>>
PARTITION of <<Partitioned Table>>
FOR VALUES WITH (MODULUS <<int>>,
REMAINDER<<int>>);

Range Table Partitioning

Divide Rows into Defined Ranges

- Range Partitioning organizes rows into ranges based on a key column or columns, with boundaries specified by:
  - Lower bound (inclusive)
  - Upper bound (exclusive)

Examples:

- Partitioning by date or identifier ranges, such as:
  - Date ranges (e.g., monthly or quarterly)
  - Identifier ranges (e.g., employee ID or customer ID)

Insertion Behavior:

- Inserting EmpID = 6 into the "Employees" Partitioned Table would place it in the "Emp6_10" Partition, due to the exclusive upper bound.

```
CREATE TABLE Employees
(EmpID INT NOT NULL,
EmpName VARCHAR(25))
PARTITION BY RANGE(EmpID);

CREATE TABLE Emp1_5 PARTITION OF Employees
FOR VALUES FROM (1) TO (6);
CREATE TABLE Emp6_10 PARTITION OF Employees
FOR VALUES FROM (6) TO (11);
CREATE TABLE Emp11_15 PARTITION OF Employees
FOR VALUES FROM (11) TO (16);
```

```
CREATE Table Transactions
(TranID INT NOT NULL,
 TranDate TIMESTAMP,
 EmpID INT,
 CustID INT)
 PARTITION BY RANGE(TranDate);

 CREATE TABLE Trans_2024_01 Partition of Transactions
 FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

 CREATE TABLE Trans_2024_02 Partition of Transactions
 FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');

 CREATE TABLE Trans_2024_03 Partition of Transactions
 FOR VALUES FROM ('2024-03-01') TO ('2024-04-01');
```

List Table Partitioning

List Partitioning

- Rows are divided into partitions based on specific values in a column.
- This Partitioning scheme is useful when data can be categorized into distinct, non-overlapping sets, and those values are predictable and stable.

```
CREATE TABLE Products
(ProdID INT,
Category VARCHAR(20),
Prod_Desc VARCHAR(50)
) PARTITION BY LIST(Category);
```

```
CREATE TABLE Prod_SportGear PARTITION OF Products
FOR VALUES IN ('SportsGear');

CREATE TABLE Prod_Electronics PARTITION OF Products
FOR VALUES IN ('Electronics');

CREATE TABLE Prod_Tools PARTITION OF Products
FOR VALUES IN ('Tools');
```

Hash Table Partitioning

- **Distribute Rows Using Hash Values**
- **Hash Partitioning divides rows by:**
  - Specifying a modulus (divisor)
  - Specifying a remainder for each partition
- **Partition Assignment:**
  - Rows are assigned to partitions based on the hash value of the partition key, divided by the modulus, producing the specified remainder.
- **Ideal Use Case:**
  - When no natural partitioning method exists, Hash Partitioning evenly distributes data.
- **Important Note:**
  - Null values are always assigned to the partition with a remainder of zero.

```sql
CREATE TABLE Customers
(CustID INT,
CustomerName VARCHAR(24))
PARTITION BY HASH(CustID);
--Want 4 Partitions:
CREATE Table Cust_Part_1 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 0);
CREATE Table Cust_Part_2 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 1);
CREATE Table Cust_Part_3 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 2);
CREATE Table Cust_Part_4 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 3);
```

```sql
CREATE TABLE Customers
(CustID INT,
CustomerName VARCHAR(24))
PARTITION BY HASH(CustName);
--Want 4 Partitions:
CREATE Table Cust_Part_1 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 0);
CREATE Table Cust_Part_2 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 1);
CREATE Table Cust_Part_3 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 2);
CREATE Table Cust_Part_4 PARTITION OF Customers
FOR VALUES WITH (modulus 4, remainder 3);
```

Partition Maintenance: Adding New Partitions

Attaching a New Partition

Range Partition Scheme example

- Create new partition based on TimeStamp Column
- Using the same CREATE TABLE.....PARTITION OF....Syntax

```
CREATE TABLE Trans_2024_04 PARTITION OF Transactions
FOR VALUES FROM ('2024-04-01') TO ('2024-05-01');
```

Partition Maintenance: Adding New Partitions Alternative Methodology

Efficient Partition Attachment

To minimize locking conflicts:

- Create a new table outside the partition structure
- Attach it as a partition later using ATTACH PARTITION

This approach:

- Only requires a SHARE UPDATE EXCLUSIVE lock (less restrictive than the ACCESS EXCLUSIVE otherwise required)
- Supports concurrent operations on the partitioned table

Optimization Tips:

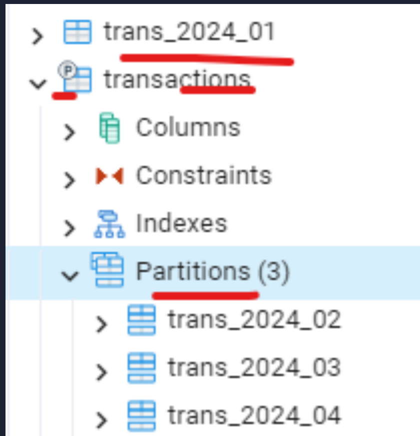
- Use CREATE TABLE … LIKE to replicate the parent table's definition
- Create a CHECK constraint enforcing partition logic **before** attachment to avoid the scan that is otherwise needed to validate the implicit partition constraint
- Drop the redundant CHECK constraint **after** the fact

```
CREATE TABLE Trans_2024_05 (LIKE Transactions INCLUDING DEFAULTS INCLUDING CONSTRAINTS);

ALTER TABLE Trans_2024_05 ADD CONSTRAINT y2024m05 CHECK (TranDate>=TIMESTAMP '2024-05-01' AND Trandate< TIMESTAMP '2024-06-01')

ALTER TABLE Transactions ATTACH PARTITION Trans_2024_05 FOR VALUES FROM ('2024-05-01') TO ('2024-06-01')

ALTER TABLE Trans_2024_05 DROP CONSTRAINT y2024m05
```

Detaching An Existing Partition

To Remove an Existing Partition from a Partitioned Table, use the DETACH PARTITION Command.

```
ALTER TABLE Transactions
DETACH PARTITION Trans_2024_01;
```



Efficiently Remove and Add Partitions

- To maintain relevant data and optimize storage
- Periodically **remove partitions** containing outdated data
- **Add new partitions** for incoming data

Key Benefit:

- Modify the partition structure to quickly eliminate large datasets
- Avoid physically moving data, reducing execution time and hassle

Optimized Data Removal:

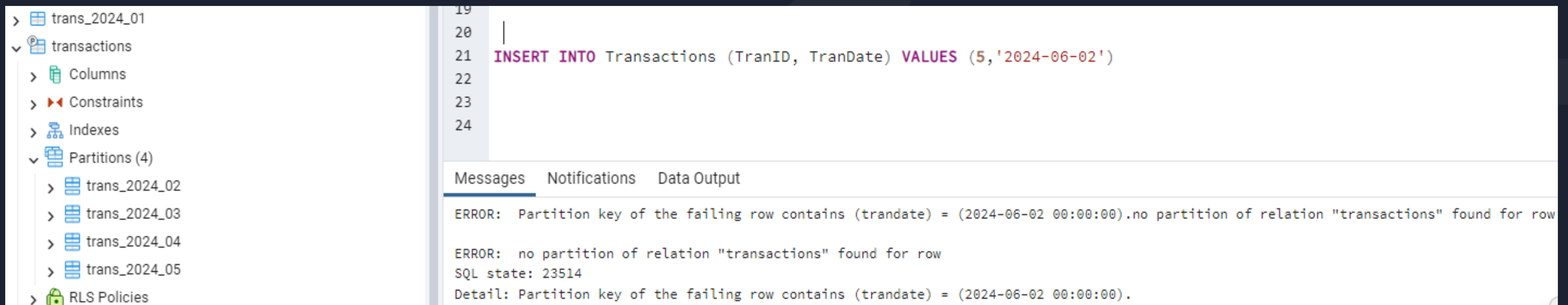
•Drop a partition table or detach and drop a partition to:

- Eliminate millions of records quickly
- Avoid ACCESS EXCLUSIVE lock on the parent table
- Outperform traditional DELETE FROM operations

Necessary Default Partition

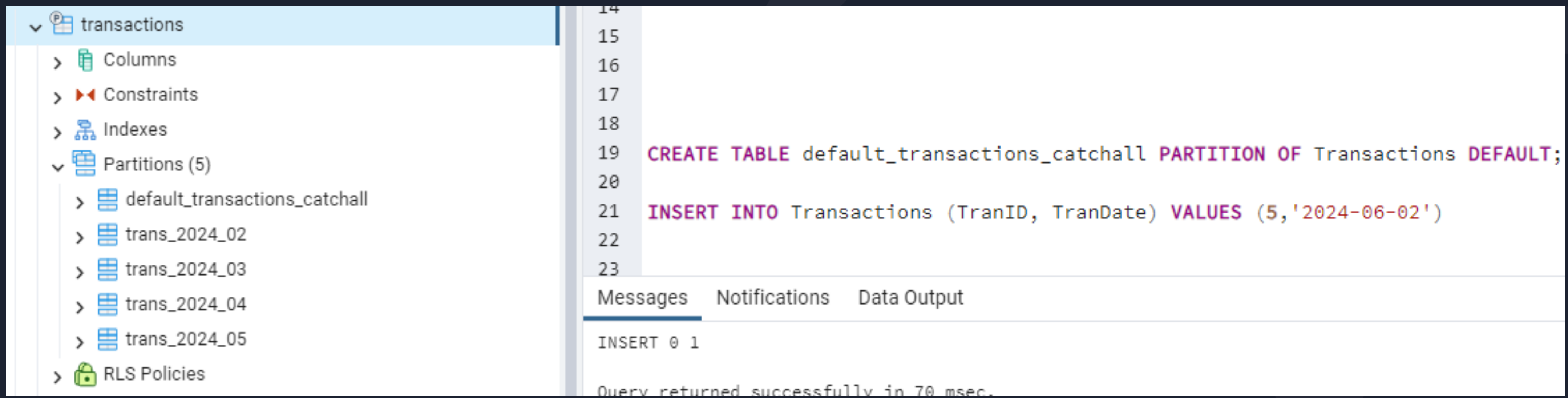
Handling Unpartitioned Records

- If a record doesn't meet the partition scheme logic:
  - Insertion will fail unless a **Default Partition** is defined
  - Default Partition acts as a catch-all for records outside existing partition boundaries



Necessary Default Partition (cont'd)

- To avoid this issue, it is recommended to create a Default Partition, where all the values that are not reflected in the mapping of the child tables will be inserted.
- These outlying rows will be caught and can be allocated to the proper partitions after the fact and migrated over as needed.



Partitioning and Tablespaces

Tablespaces allow administrators to define locations in the file system where the files representing database objects can be stored.

- Once created, a tablespace can be referred to by name when creating database objects.
- Tablespaces allow administrators to use knowledge of the usage pattern of database objects to optimize performance.
- Can place child tables on different tablespaces, affording the option of storing older, less frequently accessed data on cheaper storage media, while keeping your more frequently accessed data (like more recent timestamp-valued partition key columns) on faster storage devices.

```
CREATE TABLESPACE ts_fast location '/data/tablespaces/ts_fast';
CREATE TABLESPACE ts_just_ok location '/data/tablespaces/ts_justok';

CREATE Table Transactions2
(TranID INT NOT NULL,
 TranDate TIMESTAMP)
 PARTITION BY RANGE(TranDate);

CREATE TABLE Trans_2023 Partition of Transactions2
FOR VALUES FROM ('2023-01-01') TO ('2024-01-01')
TABLESPACE ts_just_ok;

CREATE TABLE Trans_2024 Partition of Transactions2
FOR VALUES FROM ('2024-02-01') TO ('2025-01-01');
TABLESPACE ts_fast;
```

Partition Pruning

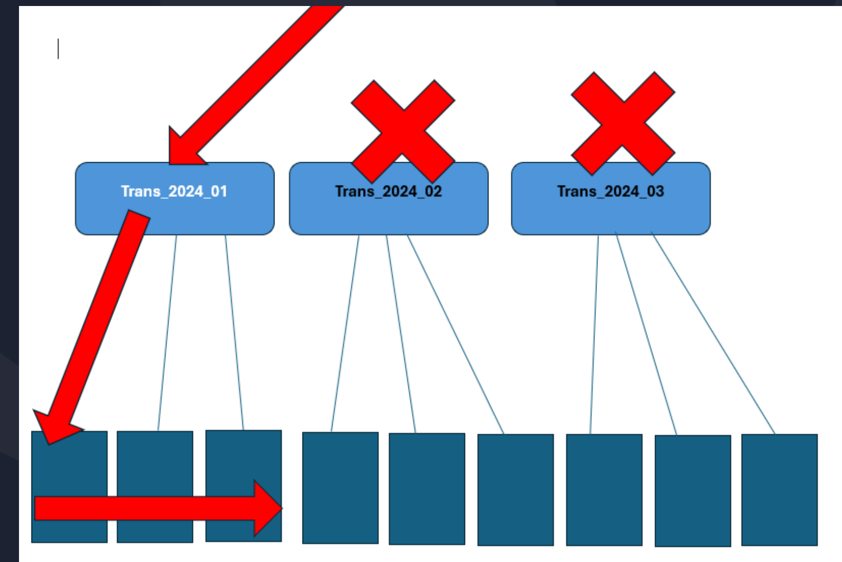
- Partition Pruning is a query optimization technique that improves performance for declaratively-partitioned tables.

- With such enabled, under certain circumstances such as when the partition key column is used in a WHERE Clause, the PostgreSQL Planner may examine the definition of each partition and prove certain partitions need not be scanned because they couldn't contain any rows meeting the query's WHERE clause by the implicit constraints put on them.

```sql
SELECT * FROM Transactions_Part
WHERE TranDate='2024-01-05'
```

sages Notifications Data Output

tranid integer	trandate timestamp without time zone	empid integer	custid integer
2	2024-01-05 00:00:00	3	5



Partition Pruning Examples

Partitioned Table

```
CREATE TABLE Transactions_Part
(TranID INT NOT NULL,
TranDate TIMESTAMP,
EmpID INT,
CustID INT)
PARTITION BY RANGE (TranDate);

CREATE TABLE Trans_2024_01 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01');

CREATE TABLE Trans_2024_02 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-02-01') TO ('2024-03-01');

CREATE TABLE Trans_2024_03 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-03-01') TO ('2024-04-01');

CREATE TABLE Trans_2024_04 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-04-01') TO ('2024-05-01');

INSERT INTO Transactions_Part
(TranID, TranDate, EmpID, CustID)
SELECT generate_series(1,1000), CURRENT_DATE-INTERVAL '3 Month',
generate_series(1,1000), generate_series(1,1000);

INSERT INTO Transactions_Part
(TranID, TranDate, EmpID, CustID)
VALUES (2,'2024-01-05',3,5);

SELECT * FROM Transactions_Part
WHERE TranDate='2024-01-05';
```

sages Notifications Data Output

tranid integer	trandate timestamp without time zone	empid integer	custid integer
2	2024-01-05 00:00:00	3	5

Partitioning Pruning Off-All Partitions are scanned.

```
1 SET enable_partition_pruning =OFF;
2 EXPLAIN SELECT * FROM Transactions_PART
3 WHERE TranDate='2024-01-05 00:00:00';
4
```

Messages Notifications Data Output

	QUERY PLAN text
1	Append (cost=0.00..113.38 rows=25 width=20)
2	-> Seq Scan on trans_2024_01 transactions_part_1 (cost=0.00..31.25 rows=8 width=...
3	Filter: (trandate = '2024-01-05 00:00:00'::timestamp without time zone)
4	-> Seq Scan on trans_2024_02 transactions_part_2 (cost=0.00..31.25 rows=8 width=...
5	Filter: (trandate = '2024-01-05 00:00:00'::timestamp without time zone)
6	-> Seq Scan on trans_2024_03 transactions_part_3 (cost=0.00..31.25 rows=8 width=...
7	Filter: (trandate = '2024-01-05 00:00:00'::timestamp without time zone)
8	-> Seq Scan on trans_2024_04 transactions_part_4 (cost=0.00..19.50 rows=1 width=...
9	Filter: (trandate = '2024-01-05 00:00:00'::timestamp without time zone)

Partitioning Pruning On-Only one Partition is scanned.

Query Query History

```
1 SET enable_partition_pruning =ON;
2 EXPLAIN SELECT * FROM Transactions_PART
3 WHERE TranDate='2024-01-05 00:00:00';
4
```

Messages Notifications Data Output

	QUERY PLAN text
1	Seq Scan on trans_2024_01 transactions_part (cost=0.00..31.25 rows=8 width=20)
2	Filter: (trandate = '2024-01-05 00:00:00'::timestamp without time zone)

Partition Pruning- Cont'd

- Note: Partition Pruning is driven only by the implicit constraints created by the Partitions, not by the presence of indexes. It is not necessary to define indexes on the key columns. Whether an index should be created for a partition depends on whether you expect that queries that scan the partition will scan a large part or just a small part, in which case an index would help.

- Additionally, partition pruning can take place both during the planning and execution phase of a query's life, such as during subquery execution or with the use of execution-time parameters like parameterized nested loop joins.

Sub-Partitioning

```
transactions_part
  Columns
  Constraints
  Indexes
  Partitions (4)
    trans_2024_01
      Constraints
      Indexes
      Partitions (2)
        trans_2024_01_1_15
        trans_2024_01_15_02_01
      RLS Policies
      Rules
      Triggers
    trans_2024_02
      Constraints
      Indexes
      Partitions (2)
        trans_2024_02_1_15
        trans_2024_02_15_03_01
      RLS Policies
      Rules
      Triggers
    trans_2024_03
      Constraints
      Indexes
      Partitions (2)
        trans_2024_03_1_15
        trans_2024_03_15_04_01
      RLS Policies
      Rules
      Triggers
    trans_2024_04
```

```sql
CREATE TABLE Transactions_Part
(TranID INT NOT NULL,
TranDate TIMESTAMP,
EmpID INT,
CustID INT)
PARTITION BY RANGE (TranDate);

CREATE TABLE Trans_2024_01 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-01-01') TO ('2024-02-01') PARTITION BY Range (TranDate);
    CREATE TABLE Trans_2024_01_1_15 PARTITION OF Trans_2024_01
    FOR VALUES FROM ('2024-01-01') TO ('2024-01-16');
    CREATE TABLE Trans_2024_01_15_02_01 PARTITION OF Trans_2024_01
    FOR VALUES FROM ('2024-01-16') TO ('2024-02-01');


CREATE TABLE Trans_2024_02 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-02-01') TO ('2024-03-01') PARTITION BY Range (TranDate);
    CREATE TABLE Trans_2024_02_1_15 PARTITION OF Trans_2024_02
    FOR VALUES FROM ('2024-02-01') TO ('2024-02-16');
    CREATE TABLE Trans_2024_02_15_03_01 PARTITION OF Trans_2024_02
    FOR VALUES FROM ('2024-02-16') TO ('2024-03-01');

CREATE TABLE Trans_2024_03 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-03-01') TO ('2024-04-01') PARTITION BY Range (TranDate);
    CREATE TABLE Trans_2024_03_1_15 PARTITION OF Trans_2024_03
    FOR VALUES FROM ('2024-03-01') TO ('2024-03-16');
    CREATE TABLE Trans_2024_03_15_04_01 PARTITION OF Trans_2024_03
    FOR VALUES FROM ('2024-03-16') TO ('2024-04-01');

CREATE TABLE Trans_2024_04 PARTITION OF Transactions_Part
FOR VALUES FROM ('2024-04-01') TO ('2024-05-01') PARTITION BY Range (TranDate);
    CREATE TABLE Trans_2024_04_1_15 PARTITION OF Trans_2024_04
    FOR VALUES FROM ('2024-04-01') TO ('2024-04-16');
    CREATE TABLE Trans_2024_04_15_05_01 PARTITION OF Trans_2024_04
    FOR VALUES FROM ('2024-04-16') TO ('2024-05-01');
```

- PostgreSQL also supports sub-partitioning, where you can further divide partitions that are expected to become larger than other partitions. This can lead to an excessive number of partitions, so restraint is advisable.
- In this example, we sub-partitioned down to days in the month, but you can choose to sub-partition on different columns and using different partition methods.

Table Partitioning and Logical Replication

- Before PostgreSQL 13, Logical Replication of Declaratively Partitioned Tables was not supported. Previously, partitions had to be replicated individually. You would have to create separate publications for each underlying “child” partition.

- With PostgreSQL 13 and beyond, a partitioned table can be published explicitly, causing all its partitions to be published automatically. Addition/removal of a partition causes it to likewise be added to or removed from the publication.

- Additionally, on the subscriber side, logical replication supports replicating into partitioned tables. Prior to this release, subscribers could only receive rows into non-partitioned tables.

Best Practices with Partitioning

- Partitioning isn't a magic bullet. If it isn't done right, with proper access pattern-awareness it can hurt overall performance in some cases, as you are now having to do more joins between disparate tables. One of the most critical design decisions is what to partition on. The best choice is generally the column or set of columns that most frequently appear in WHERE clauses.

- Choosing the Target Number of Partitions is also a critical decision. Not having enough partitions may mean that indexes remain too large and that data locality remains poor which can result in low cache hit ratios. However, having too many partitions can also lead to longer query planning times and higher memory consumption during both Planning and Execution phases.

More Reading: Where to Go From Here

Official PostgreSQL Documentation:

PostgreSQL: Documentation: 16: 5.11. Table Partitioning

https://www.postgresql.org/docs/current/ddl-partitioning.html

PgPartman Ext: An Extension to Automate the Creation and Maintenance of Table Partitions:

GitHub - pgpartman/pg_partman: Partition management extension for PostgreSQL

https://github.com/pgpartman/pg_partman

All Viscosity hosted **webinar recordings** and **conference sessions slide decks** will be posted to **OraPub.com** for free and paid members!

ORAPUB
POWERED BY VISCOSITY

Ready for some cool Oracle stuff?

It's easy to get what you want.
Just enter your current membership login, or become a free or paid member today!

- **Free tools** for the Oracle DBA
- **Free presentations** that give you great insights
- **Free public webinars** that are how-to and fun
- **How- to webinars just for paid members** and we have 80+ of them!
- **Video seminars for paid members** that go in-depth into the Oracle DB

LOGIN

NOT A MEMBER YET?

The End

I would tell you all a joke about a Partition….but I'm not sure you would get over it…..

VISCOSITY NORTH AMERICA

Data | Apps | Cloud | Infrastructure | Events | About

ENGAGE WITH US›

Keep Up to Speed with All Upcoming Events

Gain valuable skills and validate your expertise through training with Oracle and Microsoft Experts

All times listed in Central Time

Next Upcoming Event:

SQL Tuning - How Does Cardinality Work (NL vs HJ)

1 Days | 21 Hours | 43 Minutes | 48 Seconds

WEBINAR

Prepare for Oracle Database 23c

NYOUG Co-Hosted Webinar

Presented by: **Charles Kim**

February 20, 2024 11:00 AM

REGISTER NOW | Share: f t in

WEBINAR

SQL Tuning - How Does Cardinality Work (NL vs HJ)

OraPub Paid Members Only

Presented by: **Gary Gordhamer**

February 22, 2024 11:00 AM

JOIN ORAPUB | Share: f t in

WEBINAR

Observability and Management Introducing New Features in OCI Operations Insights

NYOUG Level Up Series

Presented by: **Murtaza Husain, Oracle**

February 27, 2024 11:00 AM

REGISTER NOW | Share: f t in

Keep Up To Speed With all Upcoming Events!

https://events.viscosityna.com

Follow Us Online!

Facebook.com/ViscosityNA

LinkedIn.com/company/Viscosity-North-America

@ViscosityNA

Viscosity North America

Viscosity_NA