PERCONA

Databases run better with Percona

About Me

JOBIN AUGUSTINE (47)

- Working for Percona
- Life with databases
- Started using PostgreSQL in late 90s
- Regular to pgconf India

PERCONA

Modern SQL in PostgreSQL

Features that can transform the way we develop applications.

Why to learn Modern SQL

Performance Data

ClientRead	42
WALRead	7
DataFileRead	5
DataFileFlus	
ControlFileSy	
DataFileWrite	
	.8Times ! of overall server activity, which is huge

Statement since	State since	waits
00:00:42.089291	00:00:42.080014	CPU: 0.15%, Net/Delay*: 3.49%
00:00:18.42762	00:00:18.41787	CPU: 0.55%, Net/Delay*: 60.23%
00:00:00.186576	00:00:00.186532	CPU: 0.65%, ClientRead: 0.1%, Net/Delay*: 92.50%
00:00:01.950446	00:00:01.941235	CPU: 0.55%, ClientRead: 0.1%, Net/Delay*: 75.52%
00:00:22.565522	00:00:22.554446	CPU: 0.65%, ClientRead: 0.05%, Net/Delay*: 71.87%
00:00:07.080588	00:00:07.070593	CPU: 0.1%, Net/Delay*: 9.76%
00:00:03.201683	00:00:03.201621	CPU: 0.75%, ClientRead: 0.05%, Net/Delay*: 92.06%
00:00:01.964852	00:00:01.954708	CPU: 0.7%, Net/Delay*: 64.31%
00:00:08.126933	00:00:08.117491	CPU: 0.35%, ClientRead: 0.05%, Net/Delay*: 90.30%
00:00:00.942803	00:00:00.930994	CPU: 1%, ClientRead: 0.05%, Net/Delay*: 96.91%
00:00:02.964951	00:00:02.954737	CPU: 0.3%, ClientRead: 0.05%, Net/Delay*: 73.19%
00:00:07.090723	00:00:07.078407	CPU: 0.1%, Net/Delay*: 21.48%
00:01:03.449159	00:01:03.438343	CPU: 0.05%
00:02:34.375799	00:02:34.366891	CPU: 0.1%, ClientRead: 0.1%, Net/Delay*: 16.85%
00:00:00.000063	00:00:00	CPU: 0.5%, ClientRead: 0.05%, Net/Delay*: 70.26%

Cloud Providers

DB Server Time - Wait-events, CPU time and Delays (Reference)

Event	count
[illegible]	2000
ClientRead	1592
CPU	657
XactSync	2
CPU usage is equivalent to 0.4 CPU cores (approx). Total Net/Delay* is 2.4Times ! of overall server activity. which is huge	

Statement since	State since	waits
00:00:00.000628	00:00:00.000627	ClientRead: 3.4%, CPU: 2.1%, Net/Delay*: 92.76%
00:00:00.002199	00:00:00.002198	ClientRead: 4.9%, CPU: 0.95%, Net/Delay*: 92.36%
00:00:00.00312	00:00:00.002966	ClientRead: 4.4%, CPU: 1.2%, Net/Delay*: 93.94%
00:00:00.001225	00:00:00.00108	ClientRead: 4%, CPU: 1.8%, Net/Delay*: 93.94%
00:00:00.003769	00:00:00.003619	ClientRead: 3.55%, CPU: 1.4%, Net/Delay*: 93.48%
00:00:00.003498	00:00:00.003354	ClientRead: 4.2%, CPU: 1.65%, Net/Delay*: 93.26%
00:00:00.00409	00:00:00.00202	ClientRead: 3.2%, CPU: 1.5%, Net/Delay*: 92.73%
00:00:00.000846	00:00:00.000691	ClientRead: 3.4%, CPU: 1.5%, Net/Delay*: 92.65%
00:00:00.001931	00:00:00.00178	ClientRead: 4.4%, CPU: 1.45%, Net/Delay*: 93.55%
00:00:00.004748	00:00:00.004595	ClientRead: 4.45%, CPU: 1.9%, Net/Delay*: 90.93%
00:00:00.004985	00:00:00.004812	ClientRead: 4.55%, CPU: 1.4%, Net/Delay*: 93.52%
00:00:00.003826	00:00:00.002782	ClientRead: 3.85%, CPU: 1.35%, Net/Delay*: 94.58%
00:00:00.001719	00:00:00	ClientRead: 3.65%, CPU: 1.55%, Net/Delay*: 94.22%
00:00:00.005194	00:00:00.005042	ClientRead: 3.95%, CPU: 1%, Net/Delay*: 93.96%
00:00:00.000296	00:00:00.000151	ClientRead: 2.3%, CPU: 1.35%, Net/Delay*: 86.12%

pg_gather

Measuring Network Latency

Unix tools like **ping** - ICMP

- Round Trip Latency

What is the option in restricted environments like DBaaS

Trick for DBaaS

```
pgbench -n -T 20 -f <(echo "select 'Hello World'")
```

- Create a network bound workload and measure the latency
- Need only a single connection.

```
postgres@jobin-oldlappy:~$ pgbench -T 30 -f <(echo "SELECT 'Hello World'")
pgbench (16.6)
starting vacuum...end.
```

```
postgres@jobin-oldlappy:~$ pgbench -h localhost -U postgres -T 30 -f <(echo "SELECT 'Hello World'")
Password:
pgbench (16.6)
starting vacuum...end.
transaction type: /dev/fd/63
scaling factor: 1
query mode: simple
```

```
postgres@ip-172-31-83-104:~$ PGPASSWORD=[redacted] pgbench -h [redacted] \
> -U foo -n  -T 30  -f <(echo "SELECT 'Hello World'") postgres
pgbench (16.6 (Ubuntu 16.6-1.pgdg22.04+1), server 16.4)
transaction type: /dev/fd/63
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 61910
number of failed transactions: 0 (0.000%)
latency average = 0.484 ms
initial connection time = 12.978 ms
tps = 2064.528745 (without initial connection time)
```

Network ?

It's Time to Replace TCP in the Datacenter

John Ousterhout
Stanford University

January 18, 2023

This position paper has been updated since its original publication in October of 2022 in order to correct errors and add clarification. Updates are in italics; none of the original text has been modified. The paper has triggered discussion and dissent; for pointers to comments on the paper, see the Homa Wiki: `https://homa-transport.atlassian.net/wiki/spaces/HOMA/overview#replaceTcp`.

Abstract

In spite of its long and successful history, TCP is a poor transport protocol for modern datacenters. Every significant element of TCP, from its stream orientation to its expectation of in-order packet delivery, is wrong for the datacenter. It is time to recognize that TCP's problems are too fundamental and interrelated to be fixed; the only way to harness the full performance potential of modern networks is to introduce a new transport protocol into the datacenter. Homa demonstrates that it is possible to create a transport protocol that avoids all of TCP's problems. Although Homa is not API-compatible with TCP, it should be possible to bring it into widespread usage by integrating it with RPC frameworks.

This position paper argues that TCP's challenges in the datacenter are insurmountable. Section 3 discusses each of the major design decisions in TCP and demonstrates that every one of them is wrong for the datacenter, with significant negative consequences. Some of these problems have been discussed in the past, but it is instructive to see them all together in one place. TCP's problems impact systems at multiple levels, including the network, kernel software, and applications. One example is load balancing, which is essential in datacenters in order to process high loads concurrently. Load balancing did not exist at the time TCP was designed, and TCP interferes with load balancing both in the network and in software.

Section 4 argues that TCP cannot be fixed in an evolution-

https://arxiv.org/pdf/2210.00714

Network ?

A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC

Publisher: IEEE Cite This PDF

Leah Shalev ; Hani Ayoub ; Nafea Bshara ; Erez Sabbag **All Authors**

Abstract

Document Sections

Abstract:
Amazon Web Services (AWS) took a fresh look at the network to provide consistently low latency required for supercomputing applications, while keeping the benefits of public cloud: scalability, elastic on-demand capacity, cost effectiveness, and fast adoption of newer CPUs and GPUs. We built a new network transport protocol, scalable reliable datagram (SRD), designed to utilize modern commodity multitenant datacenter networks (with a large number of network paths) while overcoming their limitations (load imbalance and inconsistent latency when unrelated flows collide). Instead of preserving packets order, SRD sends the packets over as many network paths as possible, while avoiding overloaded paths. To minimize jitter and to ensure the fastest response to network congestion fluctuations, SRD is implemented in the AWS custom Nitro networking card. SRD is used by HPC/ML frameworks on EC2 hosts via AWS elastic fabric adapter kernel-bypass interface.

https://ieeexplore.ieee.org/document/9167399

Network & Latency - Summary

- Chatty applications takes more round trip - heavy penalty
- Chatty applications needs better concurrency control and isolation levels - Poor scalability and Performance
- Performance of Network bound workloads are questionable and it is becoming worse

Other problems of poor SQL

Plan Stability Problems

- Different plan for same SQL in different environments
- Plan drifts over a time.

Wastage of resources

DB Name	Avg.Commits	Avg.Rollbacks	Avg.DMLs	Cache hit ratio	Avg.Temp Files	Avg.Temp Bytes	DB size	Age
[illegible]	20378	0	0	93	0	0	**7942959**	106188466

DB Name	Avg.Commits	Avg.Rollbacks	Avg.DMLs	Cache hit ratio	Avg.Temp Files	Avg.Temp Bytes	DB size	Age
[illegible]	18471	0	0	97	0	0	**8155619**	155045556
[illegible]	0	0	0		0	0	**7873039**	155045556
[illegible]	0	0	0		0	0	**7709199**	155045556
[illegible]	1612602	**1099813**	6073674	94	65	**5524221219**	**72733274595**	91255543
**Averages are Per Day. Total size of 4 DBs : 72.8GB								

Hidden dangers of duplicate key violations in PostgreSQL and how to avoid them

https://aws.amazon.com/blogs/database/hidden-dangers-of-duplicate-key-violations-in-postgresql-and-how-to-avoid-them/

PERCONA

Database side functions / procedures

- Considerable overhead compared to plain SQL
- Execution Engine and execution context
- Multiple Statements and loops
- Not atomic, Race condition

Ref : src/pl/plpgsql/src/pl_exec.c

Concurrency Problems

```
BEGIN TRANSACTION
    SQL Statement 1
    SQL Statement 2
    …
END;
```

Limitations of PostgreSQL

- join_collapse_limit - Order of join could affect the performance
- from_collapse_limit - Merging of sub-queries.
- geqo_threshold - GEQO vs Cost based optimization

Limitation of Humans

- Statements above 50 lines code should be questionable.
- High chance of bugs.
- Unmaintainable and difficult to comprehend

Temp file generation

DB Name	Avg.Commits	Avg.Rollbacks	Avg.DMLs	Cache hit ratio	Avg.Temp Files	Avg.Temp Bytes	DB size	Age
[illegible]	108617	622	0	12	1738	18478930554	409568568099	97518931

DB Name	Avg.Commits	Avg.Rollbacks	Avg.DMLs	Cache hit ratio	Avg.Temp Files	Avg.Temp Bytes	DB size	Age
[illegible]	3035	0	0	88	0	0	**8962851**	119241602
[illegible]	0	0	0		0	0	**8602115**	119241602
[illegible]	783653	17	1	99	0	0	**9241379**	119241602
[illegible]	265531	0	5	99	0	0	**9405219**	119241602
[illegible]	5429662	10	15797690	74	3641	**2346135703552**	**775711089443**	119241602
****Averages are Per Day. Total size of 5 DBs : 775.7GB**								

- Wrong Join order
- Filter and Sort operation at the last stage

Solution:
Use modern SQL

Concurrency of different PostgreSQL compatible solutions

```
select seat_row, seat_number, next_seat_number
from (
 select seat_row, seat_number, booked_by
     , lead(seat_number) over row_seats as next_seat_number
     , lead(booked_by  ) over row_seats as next_booked_by
 from seats
 window row_seats as (partition by seat_row order by seat_number)
) seats_with_next
where booked_by is null and next_booked_by is null;
```

Franck Pachot

https://dev.to/aws-heroes/optimistic-concurrency-control-alice-and-bob-couldnt-sit-together-5bh

CTE - WITH Clause - a magical remedy

- Overcome the limitations of general Window functions
- **join_collapse_limit** becomes more or less irrelevant
- Avoids needless repetition.
  - Substitutes temporary tables in many situations

    "A useful property of WITH queries is that they are normally ***evaluated only once per execution of the parent query***, even if they are referred to more than once by the parent query or sibling WITH queries. Thus, expensive calculations that are needed in multiple places can be placed within a WITH query to ***avoid redundant work.*** "

- Prevent unwanted multiple evaluations of functions with side-effects.

Limitations can be overcomed using CTE

```
WITH available_seats as (select ctid,seat_row, seat_number, next_seat_number
from (
 select ctid,seat_row, seat_number, booked_by
     , lead(seat_number) over row_seats as next_seat_number
     , lead(booked_by  ) over row_seats as next_booked_by
 from seats
 window row_seats as (partition by seat_row order by seat_number)
) seats_with_next
where booked_by is null and next_booked_by is null)
SELECT * FROM seats JOIN available_seats ON seats.ctid =
available_seats.ctid FOR UPDATE SKIP LOCKED;
```

FILTER clause

- SQL:2003 introduced the FILTER clause

```
SELECT  count(*) FILTER (WHERE sex = 'M') Males,
      count(*) FILTER (WHERE sex = 'F') Female,
      count(*) AS Total
FROM emp;
```

LATERAL join

- A lateral join is essentially a **foreach loop in SQL**

```
SELECT *
 a
---
 1
 2
 3
(3 rows)
```

```
 SELECT * FROM t
   JOIN LATERAL generate_series(1,a) ON TRUE;
 a | generate_series
---+-----------------
 1 |               1
 2 |               1
 2 |               2
 3 |               1
 3 |               2
 3 |               3
(6 rows)
```

SQL and Concurrency

Example (Hypothetical)

```
CREATE TABLE t1 (
   id INT PRIMARY KEY,
   data TEXT
);
```

- id need to be generated
- prefer to avoid gaps in ids
- prefer generic solution than just integer
- shouldn't be big performance sacrifice

Nextval vs Max(col)+1

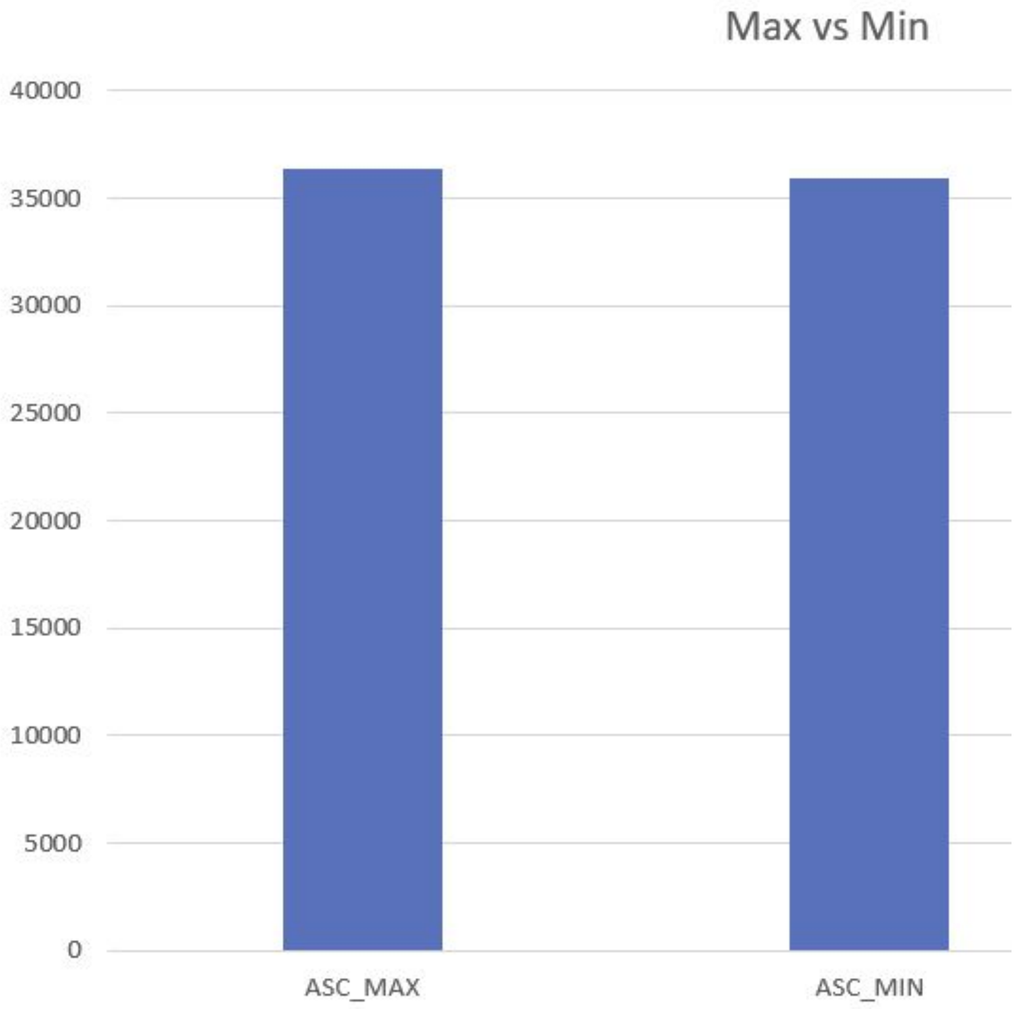
```
SELECT max(id)+1 FROM t1 ;

vs

SELECT nextval('myserial');
```

Nextval	Max()+1
36794.71	36348.21
36930.05	37203.19
36885.25	37349.97
35626.29	36465.82
36878.29	37369.66
36969.52	36929.68
37102.56	36499.59
36940.23	37196.41
36019.12	37316.14
35070.31	36467.57
Avg. QPS → **36521.63**	**36914.63**

max - execution plan

```
 Result  (cost=0.59..0.60 rows=1 width=4) (actual time=0.068..0.068 rows=1 loops=1)
   Output: $0
   Buffers: shared hit=6
   InitPlan 1 (returns $0)
         ->  Limit  (cost=0.56..0.59 rows=1 width=4) (actual time=0.061..0.062 rows=1 loops=1)
         Output: pgbench_accounts.aid
         Buffers: shared hit=6
         ->  Index Only Scan Backward using pgbench_accounts_pkey on public.pgbench_accounts  (cost=0.56..1426381.90 rows=50087962 width=4) (actual time=0.060..0.060 rows=1 loops=1)
         Output: pgbench_accounts.aid
         Index Cond: (pgbench_accounts.aid IS NOT NULL)
         Heap Fetches: 1
         Buffers: shared hit=6
 Planning Time: 0.151 ms
 Execution Time: 0.096 ms
(14 rows)
```

ASC Index and Min vs Max aggregate



- Difference in performance is not observable.
- On tests, it takes 0.2 ms of single core, including statement processing

Problem is …

```sql
INSERT INTO t1
SELECT (SELECT max(id)+1 AS newid FROM t1),'New data' RETURNING id;
```

```sql
WITH newid AS (select max(id)+1 AS newid from t1)
INSERT INTO t1 SELECT newid.newid,'New data' FROM newid RETURNING id;
```

Conflicts on concurrency

Advantages vs disadvantages

- Heights of optimism (low concurrency)
- Eliminates limitations of sequences
  - Easy data migrations
  - Different data types
  - Different logic possible
- Suitable for less concurrency

Transaction Aborts/Rollbacks are expensive

- SQL / DML Processing is pretty much completed
- Datafile (fork) writes done, Creating bloat on rollback/abort
  - Table fragmentation and performance degradation
  - Additional work for Autovacuum
- Generates WAL
  - Backups
  - Replication
- Floods Log files

Summary: Wastes all server resources, CPU, Memory, IO, Network etc

Explicit conflict handling

```
WITH maxid AS (SELECT id FROM t1 WHERE id = (SELECT max(id) FROM t1)
FOR UPDATE SKIP LOCKED)
INSERT INTO t1 SELECT maxid.id+1,'New data' FROM maxid WHERE id IS NOT
NULL RETURNING id;
```

```
 Insert on t1  (cost=9.20..9.22 rows=1 width=36) (actual time=0.057..0.060 rows=1 loops=1)
   CTE maxid
     ->  LockRows  (cost=1.17..9.20 rows=1 width=10) (actual time=0.042..0.044 rows=1 loops=1)
           InitPlan 2 (returns $1)
             ->  Result  (cost=0.60..0.61 rows=1 width=4) (actual time=0.029..0.029 rows=1 loops=1)
                   InitPlan 1 (returns $0)
                     ->  Limit  (cost=0.57..0.60 rows=1 width=4) (actual time=0.027..0.027 rows=1
loops=1)
                           ->  Index Only Scan Backward using t1_pkey on t1 t1_1  (cost=0.57..2284485.27
rows=80168040 width=4) (actual time=0.026..0.026 rows=1 loops=1)
                                 Index Cond: (id IS NOT NULL)
                                 Heap Fetches: 1
           ->  Index Scan using t1_pkey on t1 t1_2  (cost=0.57..8.59 rows=1 width=10) (actual
time=0.036..0.037 rows=1 loops=1)
                 Index Cond: (id = $1)
   ->  CTE Scan on maxid  (cost=0.00..0.02 rows=1 width=36) (actual time=0.046..0.048 rows=1 loops=1)
         Filter: (id IS NOT NULL)
 Planning Time: 0.168 ms
 Execution Time: 0.089 ms
```

Notes on Attempt 2

- Verification of data, Generation/Prepreation of data, Concurrency control can be clubbed into CTE
- There are cases where locks for synchronization are unavoidable.
- **Momentarily Row locks** with small scopes are possible.
- Instantaneous release of locks **improves the concurrency**
- Considerably **reduces the conflicts and transaction aborts**

Attempt 3 improve concurrency

- ON CONFLICT - Introduced in PostgreSQL 9.5
    - “speculative insertion”
    - first does a pre-check for existing tuples and then attempts an insert
    - No file write / bloat on conflict
    - No WAL generation on conflict

```
WITH maxid AS (SELECT max(id) AS id FROM t1)
INSERT INTO t1 SELECT maxid.id+1,'New data' FROM maxid
ON CONFLICT DO NOTHING RETURNING id;
```

ON CONFLICT

Update with Excluded data

```
…
ON CONFLICT (id) DO UPDATE SET data = EXCLUDED.data RETURNING id;
```

Specify constraint name

```
ON CONFLICT ON CONSTRAINT t1_pk
 DO UPDATE SET data = EXCLUDED.data RETURNING id;
```

WHERE clause for verifying the data

```
INSERT INTO stock VALUES (2,'Books',1,true)
ON CONFLICT (id)
DO UPDATE SET cnt = stock.cnt + EXCLUDED.cnt WHERE stock.active AND
stock.cnt < 5 RETURNING *;
```

* No record will be updated if verification fails. and returns no records

MERGE

- PostgreSQL 15+
- INSERT + UPDATE + DELETE
- Multiple WHEN clauses, with Conditions
  - Evaluated in the order specified
- For each row, match is checked and associated actions are performed.

- Complex business logic with associated data can be send to server as a capsule.
- Best for data reconciliation use cases.

Usage of MERGE

- Data and business logic can be send to server as a capsule

```
MERGE INTO tags USING (VALUES
    ('A', false),
    ('B', true),
    ('C', false),
    ('D', true)
) AS t(name, deleted)
   ON t.name = tags.name
   WHEN MATCHED AND deleted THEN DELETE
   WHEN MATCHED AND NOT deleted THEN DO NOTHING

   …
   …
```

https://hakibenita.com/postgresql-get-or-create

Usage of MERGE

- Data reconciliation use cases.
- Logical data synchronization
- Data movement to Warehouses, Reporting systems
- The ability to specify conditions rather than relying on a constraint, unlike INSERT ON CONFLICT
- Replaces stored procedures / functions in many usecases.

MERGE in PG 17

- RETURNING clause
- merge_action()

```
MERGE INTO products p
  USING stock s ON p.product_id = s.product_id
  WHEN MATCHED AND s.quantity > 0 THEN
    UPDATE SET in_stock = true, quantity = s.quantity
  WHEN MATCHED THEN
    UPDATE SET in_stock = false, quantity = 0
  WHEN NOT MATCHED THEN
    INSERT (product_id, in_stock, quantity)
      VALUES (s.product_id, true, s.quantity)
  RETURNING merge_action(), p.*;

 merge_action | product_id | in_stock | quantity
--------------+------------+----------+----------
 UPDATE       |       1001 | t        |       50
 UPDATE       |       1002 | f        |        0
 INSERT       |       1003 | t        |       10
```


PERCONA

Haki Benita   About   Subscribe

05 AUGUST 2024 / POSTGRESQL, SQL

How to Get or Create in PostgreSQL

And why it is so easy to get wrong...

APPROACH	IDEMPOTENT	CONCURRENCY	BLOAT	CONSTRAINT
INSERT	❌	-	✅	❌
SELECT INSERT	✔	❌	✅	-
SELECT INSERT SELECT	✔	❌	✅	❌
INSERT EXCEPT SELECT	✔	✔	❌	❌
INSERT WHERE NOT EXISTS	✔	❌	✅	✅
INSERT ON CONFLICT DO NOTHING	✔	✔	✅	❌
INSERT ON CONFLICT DO UPDATE	✅	✅	❌	❌
MERGE RETURNING (PostgreSQL 17+)	✔	✔	✅	✅
INSERT ON CONFLICT DO NOTHING COMMIT SELECT	✅	✅	✅	❌

✔ Not safe under the situations described in this section but otherwise OK.

https://hakibenita.com/postgresql-get-or-create

Examples

Replicating to DSS system

```
MERGE INTO emp t
 USING (SELECT * FROM remote.emp WHERE last_modified > '2025-03-01 01:48:18' ) s
 ON t.id = s.id
WHEN MATCHED THEN
    UPDATE SET name = s.name, last_modified = s.last_modified
WHEN NOT MATCHED THEN
    INSERT (id, name, sex, last_modified)
    VALUES (s.id, s.name,s.sex,s.last_modified);
```

- Source of data can be SQL query to remote database using FDW
- Water mark method of replicating data to DSS/OLAP
- Conflict resolution is clear and explicit

UNNEST

```
postgres=# SELECT * FROM unnest('{1, 2, 3, 4}'::int[],'{abc, def, hig,
klmno}'::text[]) AS value;
 unnest | unnest
--------+--------
      1 | abc
```

```
SELECT  UNNEST(ARRAY ['Collected At','Collected By','PG build', 'Last Startup','In recovery?','Cl
        UNNEST(ARRAY [CONCAT(collect_ts::text,' (',TZ.val,')'),usr,ver, pg_start_ts::text ||' ('
        pg_snapshot_xmax(snapshot)::text,pg_snapshot_xmin(snapshot)::text,current_wal::text,timel
        'ID: ' || systemid || ' Since: ' || to_timestamp ( systemid >> 32 ) || ' ('|| collect_ts-
FROM pg_gather LEFT JOIN TZ ON TRUE
```

https://www.timescale.com/blog/boosting-postgres-insert-performance/

INSERT .. UNNEST is 2.13x faster than INSERT .. VALUES at at batch size of 1000!

Additionally, The data validation and transformation possible (than COPY)

INSERT INTO ON CONFLICT

insert into emp VALUES (1,'Jobin augustine') ON CONFLICT (id) DO UPDATE SET name = EXCLUDED.name;

- A virtual “EXCLUDED” table is provided to reference the values which are rejected.

Example 1 : DML to multiple tables as a single statement

```
WITH neworder (order_description) AS (VALUES ('Order for office chairs')),
     neworderdtls (orderitem_id,nos) AS (VALUES (100,2),(101,4),(105,10)),
  head AS (INSERT INTO orderhead (order_description)
     SELECT order_description FROM neworder WHERE NOT EXISTS
      (SELECT * FROM orderhead o WHERE o.order_description = neworder.order_description)
      RETURNING order_id),
  details AS (INSERT INTO orderdtls (line_no,order_id,orderitem_id,nos)
     SELECT ROW_NUMBER() OVER () as line_no ,order_id,orderitem_id,nos
     FROM neworderdtls,head)
SELECT order_id FROM head;
```

CTE Materialization

```
WITH w AS NOT MATERIALIZED (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

Data retention. Archiving & Purging

- Old / Obsolete data need to be removed from the main database
- Preferably move the data to separate archive table on different storage
  - Make sure to delete only the data which is successfully moved to archive table
- Do it in single transaction
- Batch the operation using criteria

```
WITH rows AS (DELETE FROM pgbench_accounts WHERE aid < 10000 RETURNING *)
INSERT INTO pgbench_accountsnew SELECT * FROM rows;
```

Helps in Partitioning the tables or repairing the partitions

SQL statement

- Single Statement is **atomic**, even though it contains multiple sub statements, Even if not specified in an explicit transaction.
- Everything in a statement is **executed with the same snapshot.**
- Effect of one statement is **not visible** on another sub statement.
- **RETURNING data is the only way** to communicate changes between different WITH sub-statements

Summary

- Study the **wait-event pattern**, understand the time and resource wastage
- Design for **higher concurrency, Less locking, Reduced handshakes**.
- Be aware about isolation levels when dealing with concurrency.
- Higher concurrency comes with higher chance of collision
  - Application to handle the exceptions
  - Application to provide the retry logic.
  - Avoid aborts and rollbacks they are very expensive.
- Invest time in **writing good SQL** before trying to tune SQL

Good References:

- **Markus Winand :** https://modern-sql.com
- **How to Get or Create in PostgreSQL:** https://hakibenita.com/postgresql-get-or-create
- **Hidden dangers of duplicate key violations in PostgreSQL and how to avoid them:** https://aws.amazon.com/blogs/database/hidden-dangers-of-duplicate-key-violations-in-postgresql-and-how-to-avoid-them/
- https://www.postgresql.org/docs/current/sql.html
- https://www.postgresql.org/docs/current/functions-aggregate.html

pg_gather : https://github.com/jobinau/pg_gather
linkedin : https://www.linkedin.com/in/jobinaugustine/

Thank You!

Anonymous functions substituted by window functions

https://fljd.in/en/2024/11/25/substituting-a-variable-in-a-sql-script/

SQL Optimization

- https://danolivo.substack.com/p/could-group-by-clause-reordering
-