



PASS
Data Community
SUMMIT 2021

PostgreSQL for the SQL Server DBA

What to know when you're called in to
help!

Ryan Booz

He/Him

Director, Developer Advocacy

Timescale



Ryan Booz

He/Him

Director, Developer Advocacy

Timescale

 @ryanbooz

 softwareandbooz.com



- Husband and father of 6 amazing kids!
- 🎵, 🐝, & ☕ = 💯
- Love data, SQL databases, and teaching others about multiple data platforms
- Pluralsight author: SQL Server Triggers and Functions

<https://github.com/ryanbooz/presentations>



Roadmap

(1)

**What is PostgreSQL
and why should you
care?**

(2)

**SQL and Code
Differences**

(3)

**Quick Primer on
Query Tuning**

(4)

Community and Help

-T2371



Ed Elliott
@EdDebug

I wrote some postgres sql, apart from changing top to limit it was exactly the same as t-sql? (Cte, group by, window functions)

4:32pm · 30 Jul 2020 · Twitter Web App

2 Replies 1 Like



...

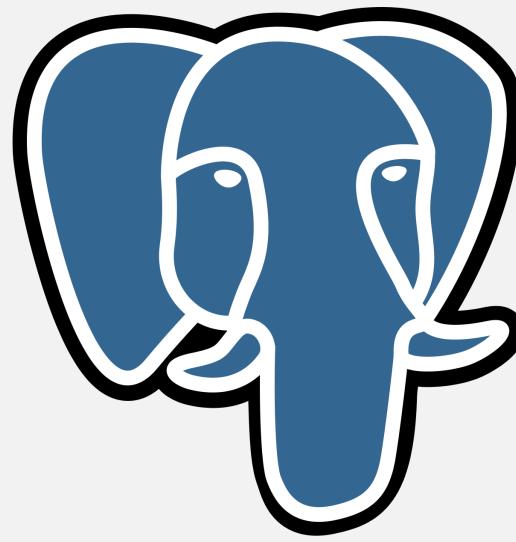


Ryan Booz @ryanbooz 30 Jul 2020

Replies to @EdDebug

What did you expect to be different?
(honestly curious)

1 reply 0 retweets 0 likes



PostgreSQL



#PASSDataCommunitySummit

PostgreSQL

- Open Source Software (OSS)
- Supports 170/177 SQL:2016 standards



#PASSDataCommunitySummit

*It's really hard to quantify this...

PostgreSQL Variants



Timescale



Why this matters to your SQL Server Career:

Introducing
Babelfish for Aurora PostgreSQL

Run SQL Server applications on PostgreSQL with little to no code changes

NEW
PREVIEW

The diagram consists of three circular icons connected by a horizontal line. The first icon contains a T-SQL logo. The second icon contains a gear and a checkmark. The third icon contains two interlocking gears.

Keep existing queries

Translation layer enables Aurora PostgreSQL to understand Microsoft SQL Server's proprietary T-SQL

Accelerate migrations

Lower risk and complete migrations faster, saving you months to years of work

Freedom to innovate

Run T-SQL code side-by-side with new open source functionality and continue developing with familiar tools

Architecture



#PASSDataCommunitySummit

Architecture

- Multi-version concurrency control (MVCC)
- Heap-based (non-Clustering)
- Relies on regular maintenance through [AUTO] VACUUM
- Statistics updated with [AUTO] ANALYZE
- Extensible architecture
- **No query hints**

Configuration

- *Every non-serverless Postgres should be tuned*
- Edit `postgresql.conf`

Helpful Tuning Resources

https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server

<https://www.youtube.com/watch?v=IFIIXpm73qtk>

<https://postgresqlco.nf/>

Configuration – The big 4

shared_buffers:

- data cache
- at least 25% of total RAM

work_mem:

- max memory for each plan operation
- JOINS, GROUP BY, ORDER BY can all have workers

maintenance_work_mem:

- memory for background tasks like VACUUM and ANALYZE

max_connections:

- exactly as it says

Extensions



Extensions

- Must be installed on server (& white-listed w/DBaaS)
- Added **per-database**
- Per-database impacts backup/restore (more later)
- Can be added to model, but never assume it exists automatically

Extensions

pg_tgram
(Trigram indexes)

PostGIS
(Geospatial)

HypoPG
(Hypothetical index)

fdw_*
(Foreign data wrapers)

CitusDB
(Horizontal clustering)

ZomboDB
(Elasticsearch index)

TimescaleDB
(Relational time-series)

tablefunc
(Pivot & crosstab)

MORE...

Extensions

See available extensions

```
SELECT * FROM pg_available_extensions ORDER BY name;
```

Install

```
CREATE EXTENSION timescaledb;
```

Update

```
ALTER EXTENSION timescaledb UPDATE;
```

Uninstall

```
DROP EXTENSION timescaledb;
```

Create/Backup/Restore



Create

- Default database = **postgres**
- Template database = **template1** ≈ **MODEL**
- “Super secret” template = **template0**
- Creating a database will use **template1** by default
 - If you want specific extensions, users, permissions, schemas – modify this template

Create database (with **template1**)

```
CREATE DATABASE myDB;
```

Backup

- **pg_basebackup for cluster-wide, PITR (with WAL backup)**
- **pg_dump** for *regular* backups
- Can be text-based (SQL script) or archive (binary)
- Only directory archives can be parallelized
- **CREATE EXTENSION** commands saved, but not versions

```
pg_dump -U postgres -Fd postgres -W -h localhost -j 2 -f testdump
```

-U = username **-Fd** = format (directory | custom | tar) **-W** = ask for password
-h = host **-j** = number of parallel jobs **-f** = filename

Restore

- **Must always** have a clean database
 - The --create option doesn't do what you think
- To restore to a new name, empty database must be created
- Only directory archives can be restored in parallel

```
| pg_restore -d test_restore -Fd testdump -U postgres -h localhost -W
```

-U = username **-Fd** = format (directory | custom | tar) **-W** = ask for password

-h = host **-d** = target database

Tooling



#PASSDataCommunitySummit



SOFTWARE AND BOOZ

shaken, not stirred

Learning, Teaching & Doing - Data, .Net, BI and more

PostgreSQL for a SQL Server DBA: The Tooling Stinks



Ryan / February 26, 2019 / PostgreSQL, SQL Server

This post is part of an impromptu series about PostgreSQL and things I am learning coming from a SQL Server background. A complete list of posts can be found on [this post](#).

UPDATE: In response to this article and much of the feedback, I've created a [second post on two other tools](#). Check that out for more context and options... and keep watching for more information on PostgreSQL tools.

So here's the deal.

Just. Stop. Looking.

About

SUBSCRIBE TO BLOG VIA EMAIL

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Email Address

SUBSCRIBE

FOLLOW ME





Rob Conery

March 4, 2019 at 6:14 pm

Hi Ryan – appreciate the write up, Postgres visual tooling is indeed far behind other tools, especially other tools that you're very used to. PgAdmin is horrendous.

Most PG DBAs that I know use psql 95% of the time and a tool like Navicat when they want to lean into the visual bits. I would offer to you that you're missing sooooo much by looking for buttons and dialog boxes when the power of psql is right there for you.

Indeed it's an investment and a mind-shift, but the ex SQL Server folks I know don't miss the visual stuff at all. I wrote a post about this, inspired by yours. Hopefully it's helpful:

<https://rob.conery.io/2019/03/04/postgresql-tools-for-the-visually-inclined/>

psql

- ≈ **mssql-cli** on steroids
- Cross-platform
- Go-to interface for 90% of community and training
- Bottom line: you have to learn at least some basic psql

Resources

- https://mydbanotebook.org/pgsql_tips_all.html
- <https://tomcam.github.io/postgres/>
- <http://postgresguide.com/utilities/pgsql.html>



- Cross-platform, Eclipse (Java) based
- Community and paid license
- Multi-database support
- Most similar feeling to SSMS thus far*



- Supports Postgres
- Notebooks work relatively well
 - MacOS Notebooks might be unstable*
- Postgres still not a first-class citizen



- Subscription fee with JetBrains
- Lots of great tooling and support
- (Almost) no-brainer if your team uses JetBrains tools
- Had some DML issues, might be fixed now



- Maintained by EDB (generally)
- Locally installed web app
- Latest = pgAdmin 4, version 6.1

SQL Differences: Queries

```
    ,x(100))  
    ,4, 'day1')  
    ,74 + 86400, 'day2')  
    ,074 + 2*86400, 'day3')  
    ,1074 + 3*86400, 'day4')  
    ,81074 + 4*86400, 'day5')  
  
    ,dateadd(second ,tim, '1/1/1970'))), * From t1  
  
DATETIME  
d  
  
me = '2009-09-02 00:57:54.000'  
me = '2009-09-03 00:57:54.000'  
  
t  
  
. -4, (dateadd(second ,tim, '1/1/1970'))) >= @StartTime  
(hour,-4, (dateadd(second ,tim, '1/1/1970'))) <= @EndTime
```

Data Type Differences

SQL Server	PostgreSQL
BIT	BOOLEAN
VARCHAR/NVARCHAR	TEXT
DATETIME/DATETIME2	TIMESTAMP
DATETIMEOFFSET	TIMESTAMP WITH TIME ZONE
UUID	CHAR(16), uuid-ossp extension, PG14
VARBINARY	BYTEA
	ARRAYS

Case Rules

- Case sensitive string comparison (LIKE/ILIKE)
- All object names are coerced to lower case
- Double quote to escape names
- Prefer lower_snake_case

Case Rules

SQL Server

```
SELECT [Name]  
      FROM [Purchasing].[Vendor]  
     WHERE BusinessEntityID = 1492
```

Postgres

```
SELECT "Name"  
      FROM "Purchasing"."Vendor"  
     WHERE "BusinessEntityID" = 1492;
```

Case Rules

SQL Server

```
SELECT [Name]  
      FROM [Purchasing].[Vendor]  
     WHERE BusinessEntityID = 1492
```

Postgres

```
SELECT name  
      FROM purchasing.vendor  
     WHERE business_entity_id = 1492;
```

LIMIT ≈ TOP

SQL Server

```
SELECT TOP(10) *
FROM [Purchasing].[Vendor]
```

Postgres

```
SELECT *
FROM purchasing.vendor
LIMIT 10;
```

LIMIT/OFFSET ≈ OFFSET/FETCH

SQL Server

```
SELECT *
    FROM [Purchasing].[Vendor]
ORDER BY name
OFFSET 10 ROWS
FETCH 10 ROWS ONLY
```

Postgres

```
SELECT *
    FROM purchasing.vendor
ORDER BY name
LIMIT 10 OFFSET 10;
```

ORDER/GROUP BY

SQL Server

```
SELECT SUBSTRING(name,1,1) nameCohort, SUM(amount) totalSales  
    FROM [Purchasing].[Sales]  
GROUP BY SUBSTRING(name,1,1)  
ORDER BY nameCohort, SUM(amount)
```

Postgres

```
SELECT SUBSTRING(name,1,1) name_cohort, SUM(amount) total_sales  
    FROM purchasing.sales  
GROUP BY name_cohort  
ORDER BY name_cohort, total_sales;
```

ORDER/GROUP BY

SQL Server

```
SELECT SUBSTRING(name,1,1) nameCohort, SUM(amount) totalSales  
FROM [Purchasing].[Sales]  
GROUP BY SUBSTRING(name,1,1)  
ORDER BY nameCohort, SUM(amount)
```

Postgres

```
SELECT SUBSTRING(name,1,1) name_cohort, SUM(amount) total_sales  
FROM purchasing.sales  
GROUP BY 1  
ORDER BY 1, 2;
```

DAY/MONTH/YEAR() ≈ DATE_PART()

SQL Server

```
SELECT COUNT(*), YEAR(OrderDate)
FROM Sales.Invoices
GROUP BY YEAR(OrderDate)
ORDER BY YEAR(OrderDate) DESC
```

Postgres

```
SELECT COUNT(*), DATE_PART('year',order_date) order_year
FROM sales.invoices
GROUP BY order_year
ORDER BY order_year DESC;
```

GETDATE() ≈ NOW()

SQL Server

```
SELECT GETDATE();
```

Postgres

```
SELECT now();
```

Date Math

SQL Server

```
SELECT COUNT(*), YEAR(OrderDate)
FROM Sales.Invoices
WHERE OrderDate > DATEADD(MONTH,-1,GETDATE())
GROUP BY YEAR(OrderDate)
```

Postgres

```
SELECT COUNT(*), DATE_PART('year',order_date) order_year
FROM sales.invoices
WHERE order_date > NOW() - INTERVAL '1 MONTH'
GROUP BY order_year;
```

Generate Fake Data

- SQL Server has no generic function for generating tables of data quickly
- Many examples online using recursive CTE's, base tables with window functions, CROSS JOIN
- `generate_series()` is a built-in Postgres function for generating series of numbers and dates

Generate Data

SQL Server

```
WITH r AS (
    SELECT 1 AS n
    UNION ALL
    SELECT n+1 FROM r WHERE n+1<=100
)
SELECT * FROM r
```

Generate Data

Postgres

```
SELECT * FROM generate_series(1,100);
```

```
SELECT * FROM generate_series(0,100,5);
```

```
SELECT * FROM  
generate_series('2021-08-01','2021-08-24',INTERVAL '1 hour');
```

SQL Differences: Code



```
    "MIRROR_X":  
        use_x = True  
        use_y = False  
        mod.use_z = False  
    condition == "MIRROR_Y":  
        mod.use_x = False  
        mod.use_y = True  
        mod.use_z = False  
    condition == "MIRROR_Z":  
        mirror_mod.use_x = False  
        mirror_mod.use_y = False  
        mirror_mod.use_z = True  
  
    selection at the end -add  
    mirror_ob.select= 1  
    mirror_mod.select=1  
    context.scene.objects.active  
    ("Selected" + str(modifier))  
    mirror_ob.select = 0  
    bpy.context.selected_objects  
    data.objects[one.name].se  
  
    print("please select exactly one object")  
    --- OPERATOR CLASSES ---  
  
    types.Operator):  
        @n X mirror to the selected  
        object.mirror_mirror_x"  
        "mirror X"  
        context):  
        "context.active_object is not
```

Anonymous Code blocks

- Default query language is SQL
- Does not support anonymous code blocks
 - Variables
 - Loops
 - T-SQL like features
- Requires DO blocks, Functions, or SPROCs
 - Defaults to pl/pgsql language ≈ T-SQL

Inline T-SQL vs pl/pgsql

SQL Server

```
DECLARE @PurchaseName AS NVARCHAR(50)  
SELECT @PurchaseName = [Name]  
    FROM [Purchasing].[Vendor]  
 WHERE BusinessEntityID = 1492  
PRINT @PurchaseName
```

Inline T-SQL vs pl/pgsql

Postgres

```
DO $$  
  DECLARE purchase_name TEXT;  
  
BEGIN  
  
  SELECT name  
  FROM purchasing.vendor  
  WHERE business_entity_id = 1492 INTO purchase_name;  
  
  RAISE NOTICE '%', purchase_name;  
  
END;
```

LATERAL ≈ APPLY

- CROSS/OUTER APPLY is an extremely helpful optimization fence in T-SQL
 - Often used as an “optimization fence”
 - Outer query iterates an inner query
 - Helpful when JOIN isn’t possible
 - Retrieve top ordered row for each item in a list (ie. “most recent value for each item”)
 - Iterating a function with value from outer query
- LATERAL JOIN allows the same functionality

LATERAL ≈ APPLY

SQL Server

```
SELECT id, name, dayMonth, total
      FROM Vendor v1
CROSS APPLY (
    SELECT TOP(1) dayMonth, SUM(amount) total
      FROM Sales
     WHERE customerID = v1.id
   GROUP BY dayMonth
  ORDER BY SUM(amount) DESC
) s1
```

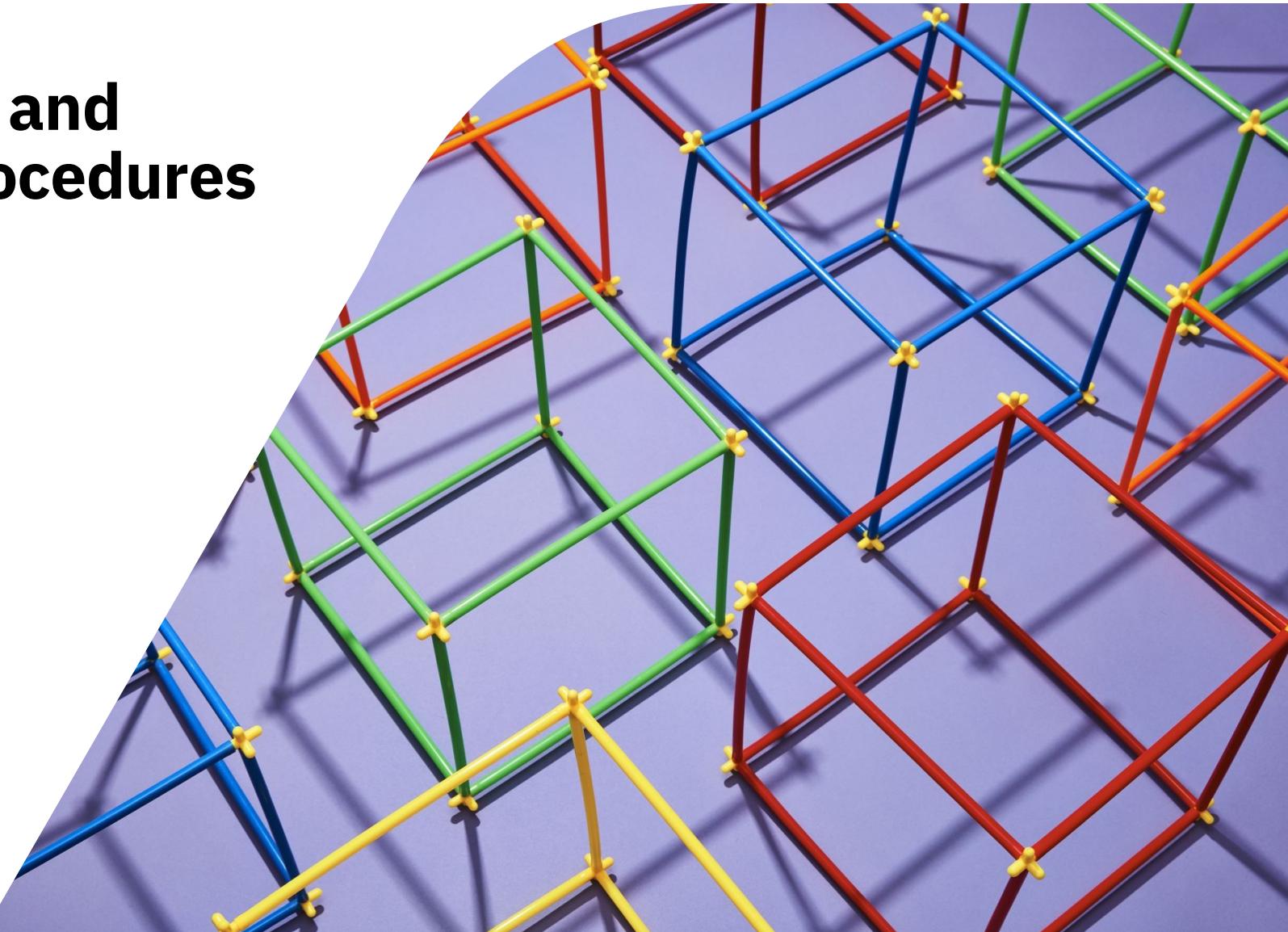
LATERAL ≈ APPLY

Postgres

```
SELECT id, name, day_month, total
  FROM vendor v1
INNER JOIN LATERAL (
    SELECT day_month, SUM(amount) total
      FROM sales
     WHERE customer_id = v1.id
   GROUP BY day_month
  ORDER BY total DESC
  LIMIT 1
) on true s1;
```

Functions and Stored Procedures

#PASSDataCommunitySummit



Functions

SQL Server

- Scaler
- Inline Table Valued
- Multi-statement Table Valued
- No tuning hints for query planner
- TSQL Only

Postgres

- Scaler
- Table/Set Types
- Triggers
- Query planner tuning parameters
- Any procedural language, but typically PL/pgSQL

Stored Procedures

SQL Server

- Complex logic
- Multi-language
- Contained in calling transaction scope
- With some work, can return results

Postgres

- Complex logic
- Multi-language
- Can issue COMMIT/ROLLBACK and keep processing
- Can return a single `INOUT` value
- Introduced in PostgreSQL 11

Triggers



Triggers

- Functions applied to tables
- INSERT/UPDATE/DELETE
- BEFORE/AFTER/INSTEAD OF
- ROW and STATEMENT
- Conditional
- NEW and OLD internal variables
- ...Reusable across tables!

Create Trigger Function

Postgres

```
CREATE OR REPLACE FUNCTION user_log()
RETURNS TRIGGER AS
$BODY$
BEGIN
    INSERT INTO UserLog (Username, Message) VALUES (NEW.Username, NEW.Message)
END;
$BODY$
LANGUAGE plpgsql;
```

Apply Trigger

Postgres

```
CREATE TRIGGER tu_users
AFTER UPDATE
ON Users
FOR EACH ROW
WHEN (OLD.FirstName IS DISTINCT FROM NEW.FirstName)
EXECUTE PROCEDURE user_log();
```

Query Tuning with EXPLAIN and pg_stat_statements

#PASSDataCommunitySummit



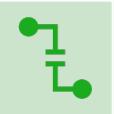
EXPLAIN in Practice

- EXPLAIN = Estimated plan
- EXPLAIN ANALYZE = Actual plan
- EXPLAIN (ANALYZE,BUFFERS)
 - Query plan with disk IO
- EXPLAIN (ANALYZE,BUFFERS,VERBOSE)
 - Additional details on columns, schemas, etc.

EXPLAIN

- There is no built-in visual execution plan
- EXPLAIN provides textual plan
 - pgAdmin does attempt some visualizations
- Websites for visualizations and suggestions
 - <https://www.pgmustard.com/>
 - <https://explain.depesz.com/>

Reading EXPLAIN



--> Nodes



Read inside-out



Join/Aggregate/Merge/Append at each level

```

EXPLAIN (ANALYZE, BUFFERS)
SELECT customer_name, count(*), date_part('year',order_date)::int order_year
FROM sales.orders o
    INNER JOIN sales.customers c ON o.customer_id=c.customer_id
WHERE c.customer_id=100
GROUP BY customer_name,order_year
ORDER BY order_year DESC;

```

```

GroupAggregate (cost=1827.91..1830.76 rows=102 width=35) (actual time=11.956..13.754 rows=4 loops=1)
  Group Key: ((date_part('year')::text, (o.order_date)::timestamp without time zone))::integer, c.customer_name
  Buffers: shared hit=903
-> Sort (cost=1827.91..1828.18 rows=107 width=27) (actual time=11.700..12.686 rows=107 loops=1)
    Sort Key: ((date_part('year')::text, (o.order_date)::timestamp without time zone))::integer DESC, c.customer_name
    Sort Method: quicksort Memory: 33kB
    Buffers: shared hit=903
      -> Nested Loop (cost=0.28..1824.30 rows=107 width=27) (actual time=0.113..10.649 rows=107 loops=1)
          Buffers: shared hit=903
            -> Index Scan using pk_sales_customers on customers c (cost=0.28..2.49 rows=1 width=27)
              Index Cond: (customer_id = 100)
              Buffers: shared hit=3
            -> Seq Scan on orders o (cost=0.00..1819.94 rows=107 width=8) (actual time=0.053..8.413 rows=107 loops=1)
              Filter: (customer_id = 100)
              Rows Removed by Filter: 73488
              Buffers: shared hit=900
Planning Time: 0.267 ms
Execution Time: 13.934 ms

```

```
GroupAggregate (cost=1827.91..1830.76 rows=102 width=35) (actual time=11.956..13.754 rows=4 loops=1)
  Group Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer), c.customer_name
  Buffers: shared hit=903
-> Sort (cost=1827.91..1828.18 rows=107 width=27) (actual time=11.700..12.686 rows=107 loops=1)
    Sort Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer) DESC, c.customer_name
    Sort Method: quicksort Memory: 33kB
    Buffers: shared hit=903
-> Nested Loop (cost=0.28..1824.30 rows=107 width=27) (actual time=0.113..10.649 rows=107 loops=1)
    Buffers: shared hit=903
      -> Index Scan using pk_sales_customers on customers c (cost=0.28..2.49 rows=1 width=27)
          Index Cond: (customer_id = 100)
          Buffers: shared hit=3
      -> Seq Scan on orders o (cost=0.00..1819.94 rows=107 width=8) (actual time=0.053..8.413 rows=107 loops=1)
          Filter: (customer_id = 100)
          Rows Removed by Filter: 73488
          Buffers: shared hit=900
Planning Time: 0.267 ms
Execution Time: 13.934 ms
```

```
GroupAggregate  (cost=1827.91..1830.76 rows=102 width=35)
                (actual time=11.956..13.754 rows=4 loops=1)
Group Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer), c.customer_name
Buffers: shared hit=903
-> Sort  (cost=1827.91..1828.18 rows=107 width=27) (actual time=11.700..12.686 rows=107 loops=1)
    Sort Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer) DESC, c.customer_name
Sort Method: quicksort Memory: 33kB
Buffers: shared hit=903
-> Nested Loop  (cost=0.28..1824.30 rows=107 width=27) (actual time=0.113..10.649 rows=107 loops=1)
    Buffers: shared hit=903
        -> Index Scan using pk_sales_customers on customers c  (cost=0.28..2.49 rows=1 width=27)
            Index Cond: (customer_id = 100)
            Buffers: shared hit=3
        -> Seq Scan on orders o  (cost=0.00..1819.94 rows=107 width=8) (actual time=0.053..8.413 rows=107 loops=1)
            Filter: (customer_id = 100)
            Rows Removed by Filter: 73488
            Buffers: shared hit=900
Planning Time: 0.267 ms
Execution Time: 13.934 ms
```

explain.depesz.com

PostgreSQL's explain analyze made readable

[new explain](#) [history](#) [help](#) [about](#) [contact](#)

[login](#)

Result: TXM1

[Settings](#)

[Add optimization](#)

[HTML](#) [SOURCE](#) [STATS](#)

#	exclusive	inclusive	rows x	rows	loops	node
1.	1.164	12.898	↑ 25.5	4	1	→ GroupAggregate (cost=1,827.91..1,830.76 rows=102 width=35) (actual time=10.690..12.898 rows=4 loops=1) Group Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer), c.customer_name Buffers: shared hit=903
2.	2.829	11.734	↑ 1.0	107	1	→ Sort (cost=1,827.91..1,828.18 rows=107 width=27) (actual time=10.442..11.734 rows=107 loops=1) Sort Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer) DESC, c.customer_name Sort Method: quicksort Memory: 33kB Buffers: shared hit=903
3.	2.071	8.905	↑ 1.0	107	1	→ Nested Loop (cost=0.28..1,824.30 rows=107 width=27) (actual time=0.096..8.905 rows=107 loops=1) Buffers: shared hit=903
4.	0.044	0.044	↑ 1.0	1	1	→ Index Scan using pk_sales_customers on customers c (cost=0.28..2.49 rows=1 width=27) (actual time=0.023..0.044 rows=1 loops=1) Index Cond: (customer_id = 100) Buffers: shared hit=3
5.	6.790	6.790	↑ 1.0	107	1	→ Seq Scan on orders o (cost=0.00..1,819.94 rows=107 width=8) (actual time=0.040..6.790 rows=107 loops=1) Filter: (customer_id = 100) Rows Removed by Filter: 73,488 Buffers: shared hit=900

Planning time : 0.423 ms

Execution time : 13.140 ms



```

EXPLAIN (ANALYZE, BUFFERS)
SELECT customer_name, count(*), date_part('year',order_date)::int order_year
FROM sales.orders o
    INNER JOIN sales.customers c ON o.customer_id=c.customer_id
WHERE c.customer_id=100
GROUP BY customer_name,order_year
ORDER BY order_year DESC;

```

```

GroupAggregate (cost=119.25..122.10 rows=102 width=35) (actual time=5.175..7.109 rows=4 loops=1)
  Group Key: ((date_part('year')::text, (o.order_date)::timestamp without time zone))::integer, c.customer_name
  Buffers: shared hit=97 read=2
-> Sort (cost=119.25..119.51 rows=107 width=27) (actual time=4.922..5.864 rows=107 loops=1)
    Sort Key: ((date_part('year')::text, (o.order_date)::timestamp without time zone))::integer DESC, c.customer_name
    Sort Method: quicksort Memory: 33kB
    Buffers: shared hit=97 read=2
      -> Nested Loop (cost=2.50..115.64 rows=107 width=27) (actual time=0.199..3.903 rows=107 loops=1)
          Buffers: shared hit=97 read=2
            -> Index Scan using pk_sales_customers on customers c (cost=0.28..) (actual time=0.028..0.047 rows=1 loops=1)
              Index Cond: (customer_id = 100)
              Buffers: shared hit=3
            -> Bitmap Heap Scan on orders o (cost=2.22..111.28 rows=107 width=8) (actual time=0.127..1.585 rows=107 loops=1)
              Recheck Cond: (customer_id = 100)
              Heap Blocks: exact=94
              Buffers: shared hit=94 read=2
                -> Bitmap Index Scan on ix_orders_customer_id (cost=0.00..) (actual time=0.076..0.084 rows=107 loops=1)
                  Index Cond: (customer_id = 100)
                  Buffers: shared read=2
Planning Time: 0.653 ms
Execution Time: 7.287 ms

```



explain.depesz.com

PostgreSQL's explain analyze made readable

[new explain](#) [history](#) [help](#) [about](#) [contact](#)

[login](#)

Result: nBHu

Settings

Add optimization

#	exclusive	inclusive	rows x	rows	loops	read	node
1.	1.332	7.565	↑ 25.5	4	1	0	→ GroupAggregate (cost=119.25..122.10 rows=102 width=35) (actual time=5.339..7.565 rows=4 loops=1) Group Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer), c.customer_name Buffers: shared hit=97 read=2
2.	2.297	6.233	↑ 1.0	107	1	0	→ Sort (cost=119.25..119.51 rows=107 width=27) (actual time=5.004..6.233 rows=107 loops=1) Sort Key: ((date_part('year'::text, (o.order_date)::timestamp without time zone))::integer) DESC, c.customer_name Sort Method: quicksort Memory: 33kB Buffers: shared hit=97 read=2
3.	2.243	3.936	↑ 1.0	107	1	0	→ Nested Loop (cost=2.50..115.64 rows=107 width=27) (actual time=0.230..3.936 rows=107 loops=1) Buffers: shared hit=97 read=2
4.	0.072	0.072	↑ 1.0	1	1	0	→ Index Scan using pk_sales_customers on customers c (cost=0.28..2.49 rows=1 width=27) (actual time=0.052..0.072 rows=1 loops=1) Index Cond: (customer_id = 100) Buffers: shared hit=3
5.	1.513	1.621	↑ 1.0	107	1	0	→ Bitmap Heap Scan on orders o (cost=2.22..111.28 rows=107 width=8) (actual time=0.145..1.621 rows=107 loops=1) Recheck Cond: (customer_id = 100) Heap Blocks: exact=94 Buffers: shared hit=94 read=2
6.	0.108	0.108	↑ 1.0	107	1	16 KB	→ Bitmap Index Scan on ix_orders_customer_id (cost=0.00..2.20 rows=107 width=0) (actual time=0.100..0.108 rows=107 loops=1) Index Cond: (customer_id = 100) Buffers: shared read=2

Planning time : 0.558 ms

Execution time : 7.794 ms



PASS
Data Community
SUMMIT 2021

pg_stat_statement

- MUST HAVE Extension for query analysis
- Installed in all/most DBaaS offerings
- Does not save plans or provide recommendations
- Allows you to do triage: “Is this a big query issue, or death by 1,000 small cuts?”

pg_stat_statement: total time w/ cache hit ratio

```
SELECT query, calls, total_time, rows,
       100.0 * shared_blk_hit/nullif(shared_blk_hit + shared_blk_read, 0)
          AS hit_percent
  FROM pg_stat_statements
 ORDER BY total_time DESC
LIMIT 5;
```

```
-[ RECORD 1 ]-----
query      | SELECT * FROM sales.orders where customer_id = ?
calls       | 1000
total_time  | 2451.00000000000
Rows.       | 521
hit_percent | 99.9873421
```

*Lots of good videos/tutorials, but also [see the docs](#) for more help and ideas.

#PASSDataCommunitySummit



Community & Help

#PASSDataCommunitySummit



Community & Help

- Postgres Slack
- Twitter (#postgres, #postgresql)
- StackOverflow (dba.stackexchange.com)
- Planet Postgres (blog aggregator)
- Mailing lists (postgresql.org/list)
- PostgreSQL Docs (postgresql.org/docs)

Books and Courses

- [The Art of PostgreSQL](#)
- [PostgreSQL Query Optimization \(Apress\)](#)
- [A Curious Moon \(Rob Conery\)](#)

What Questions do you have?

Session evaluation

Your feedback is important to us



Evaluate this session at:

www.PASSDataCommunitySummit.com/evaluation



Thank you

Ryan Booz

@ryanbooz

ryan@softwareandbooz.com



PASS
Data Community
SUMMIT 2021