Postgres for the SQL Server DBA

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

Ryan Booz

Director, Developer Advocates
Timescale

@RyanBooz

www.SoftwareAndBooz.com

https://github.com/ryanbooz/presentations

About Me

- Husband and Father of 6
- Small farmette, ~25 chickens & 5 beehives
- Learned to program in high school through the Pittsburgh Supercomputing Center using a DEC 5000
- 19 years of relational DB experience – even DB2!!
- Pluralsight author:
  Programming SQL Server Triggers and Functions

Roadmap

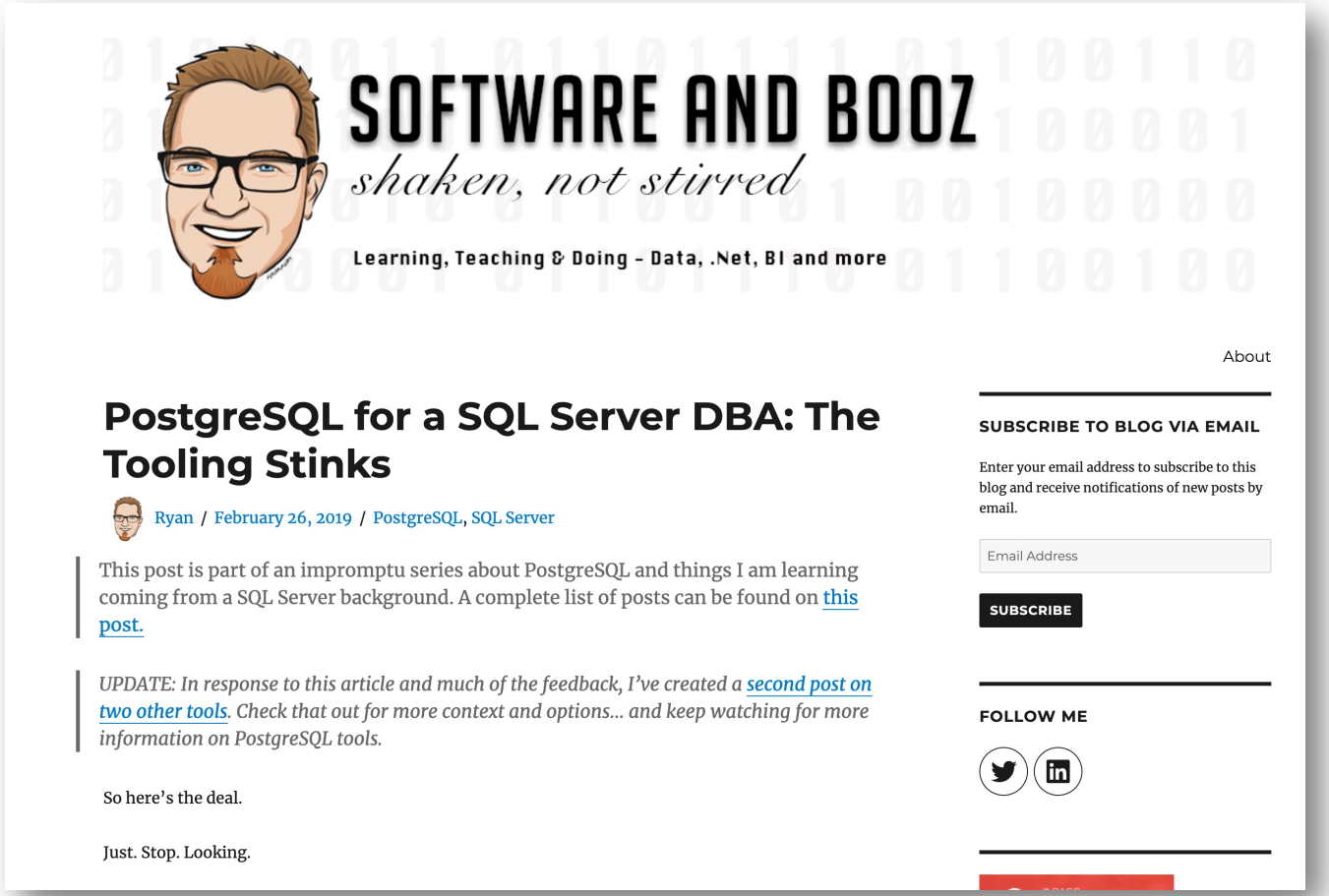
Part 1: Architecture/Setup
- What is PostgreSQL?
- Architecture
- Extensions
- Backup/Restore

Part 2: Tooling
- Tooling

Part 3: High-level SQL & Query Differences
- SQL Differences: Queries
- SQL Differences: Code
- SPROC's & Functions
- Triggers
- EXPLAIN

Part 4: Community
- Community

-T2371



SOFTWARE AND BOOZ

shaken, not stirred

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

About

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.

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 toold, 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/


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
Replying to @EdDebug
What did you expect to be different? (honestly curious)
1


Microsoft®
SQL Server®
≈

PostgreSQL

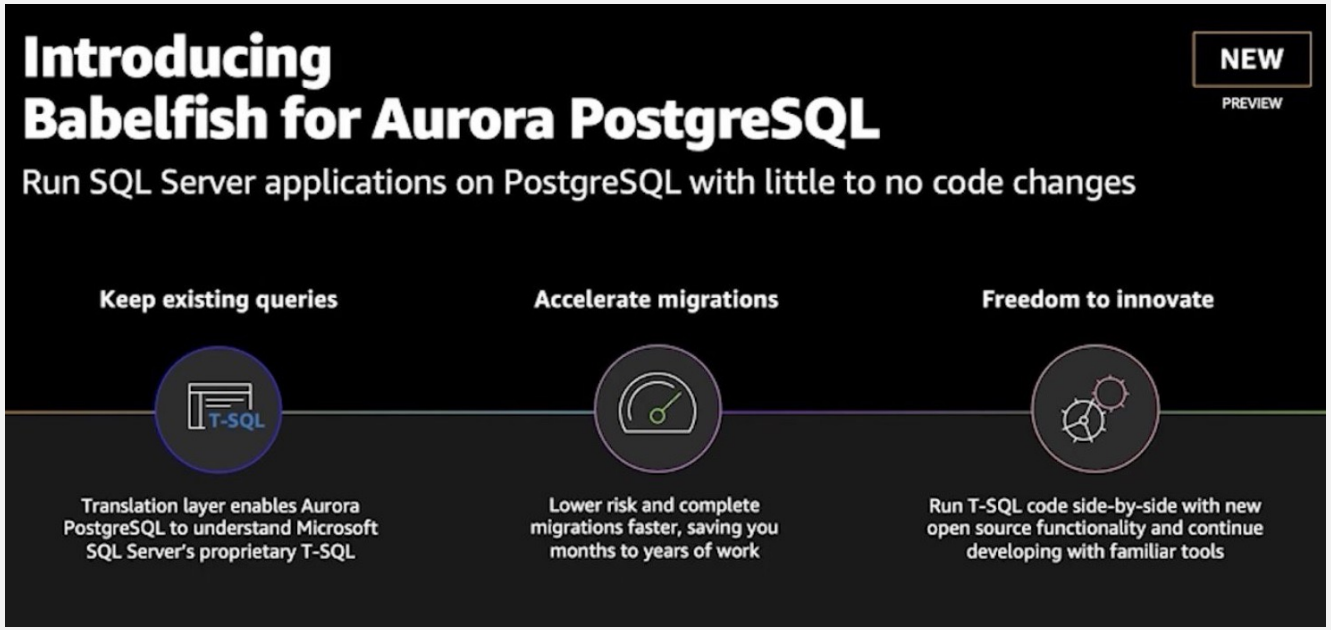
PostgreSQL

- Open Source Software (OSS)
- Maintained by core, elected team
- Upgraded through an organized CommitFest
- Starting with PostgreSQL 10, major version almost yearly
- Supports 170/177 SQL:2016 standards
- Fastest growing relational database*

*It's really hard to quantify this…

PostgreSQL Variants

PostgreSQL Variants



Architecture

Architecture

- Multi-version concurrency control (MVCC)
- Heap-based (non-Clustering)
- Relies on regular maintenance through VACUUM/AUTO VACUUM
- Statistics updated with ANALYZE
- Supports many index types (not just Btree)
- **No query hints**

Configuration

- postgresql.conf
- *Every non-serverless Postgres should be tuned*
  - Here's looking at you AWS RDS!
- Main settings
  - **shared_buffers:** data cache
  - **work_mem:** max memory for each plan operation
  - **maintenance_work_mem:** memory for background tasks
  - **max_connections:** exactly as it says
- Help:
  - https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server
  - https://www.youtube.com/watch?v=IFIXpm73qtk
  - https://postgresqlco.nf/

Extensions

Extensions

Postgres adds **per-database** functionality through extensions

- Trigram indexes
- Geography with **PostGIS**
- Foreign data wrappers for many databases (both SQL and NoSQL)
- Hypothetical indexes to test query plans
- Data analysis with PL/R & PL/Python
- Time-series data with **TimescaleDB**
- Horizontal clustering with **CitusDB**
- **tablefunc** for 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`** (kind of)
- 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
```

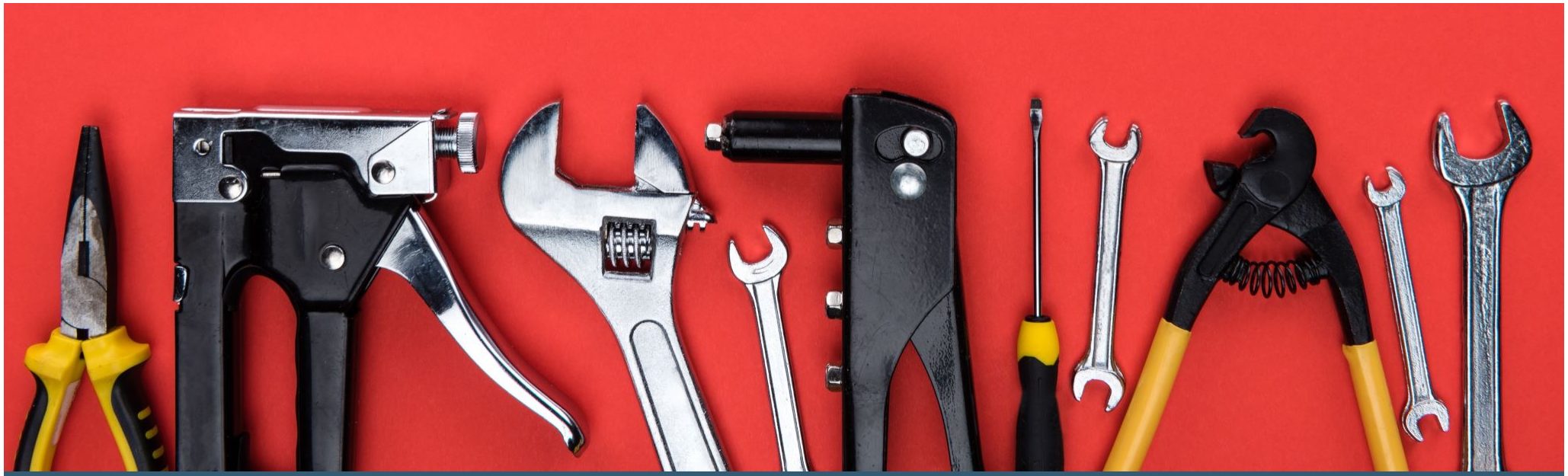
-U = username **-Fd** = format (directory | custom | tar) **-W** = ask for password
-h = host **-j** = number of parallel jobs **-f** = filename

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

psql

- ≈SQL Server 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/psql_tips_all.html
- https://tomcam.github.io/postgres/
- http://postgresguide.com/utilities/psql.html

DBeaver

- Eclipse (Java) based
- Multi-database support
- Most similar feeling to SSMS thus far*
- Cross-platform

*For me.YMMV

Azure Data Studio

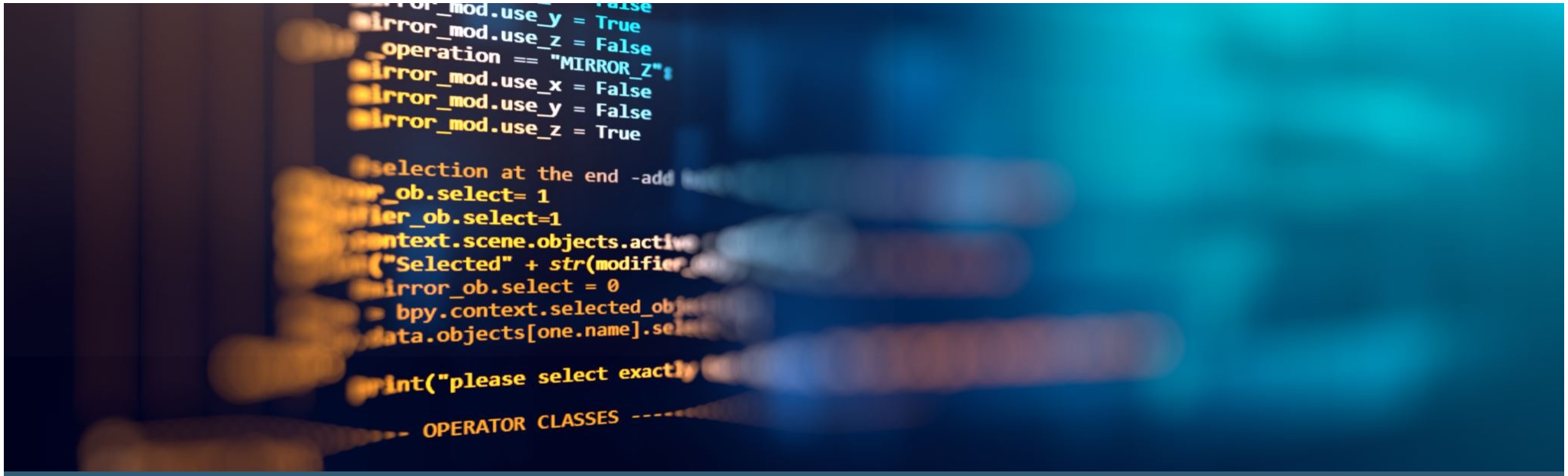
- Supports Postgres
- Notebooks work relatively well
  - At least on MacOS, Notebooks still crash regularly in my experience
- Still not a first-class citizen

Datagrip

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

pgAdmin

- Maintained by EDB (generally)
- Locally installed web app
- Latest = pgAdmin 4, version 5.3



SQL Differences: Queries

Data Type Differences

SQL Server	PostgreSQL
BIT	BOOLEAN
VARCHAR/NVARCHAR	TEXT
DATETIME/DATETIME2	TIMESTAMP
DATETIMEOFFSET	TIMESTAMP WITH TIME ZONE
UUID	CHAR(16) or **`uuid-ossp`** extension
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
```

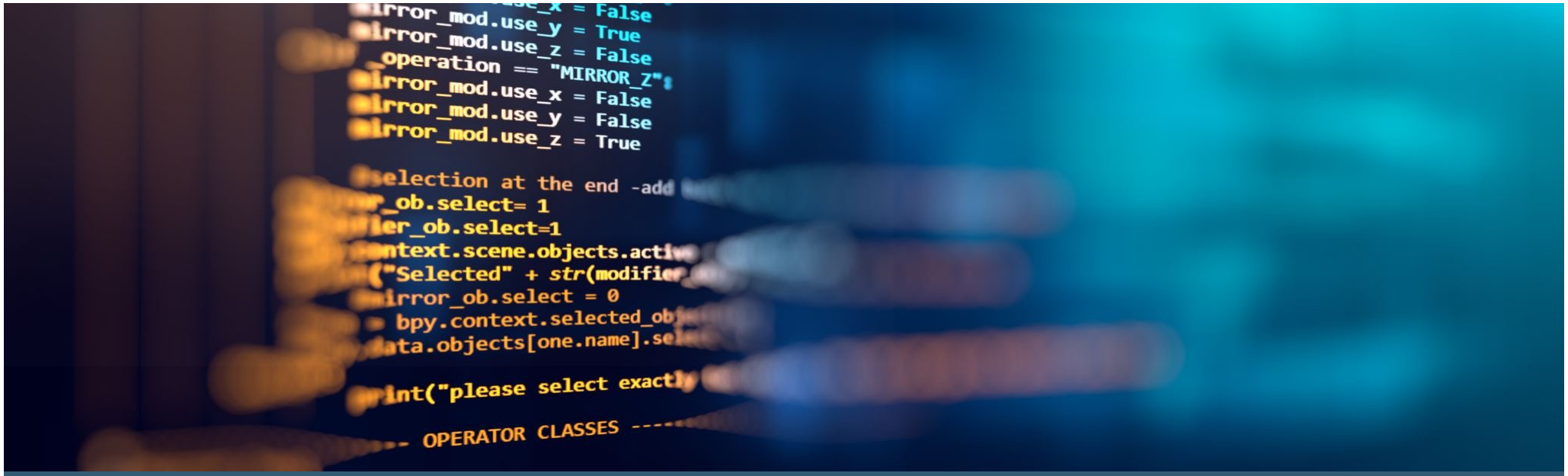
Generate Data

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

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
  - 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

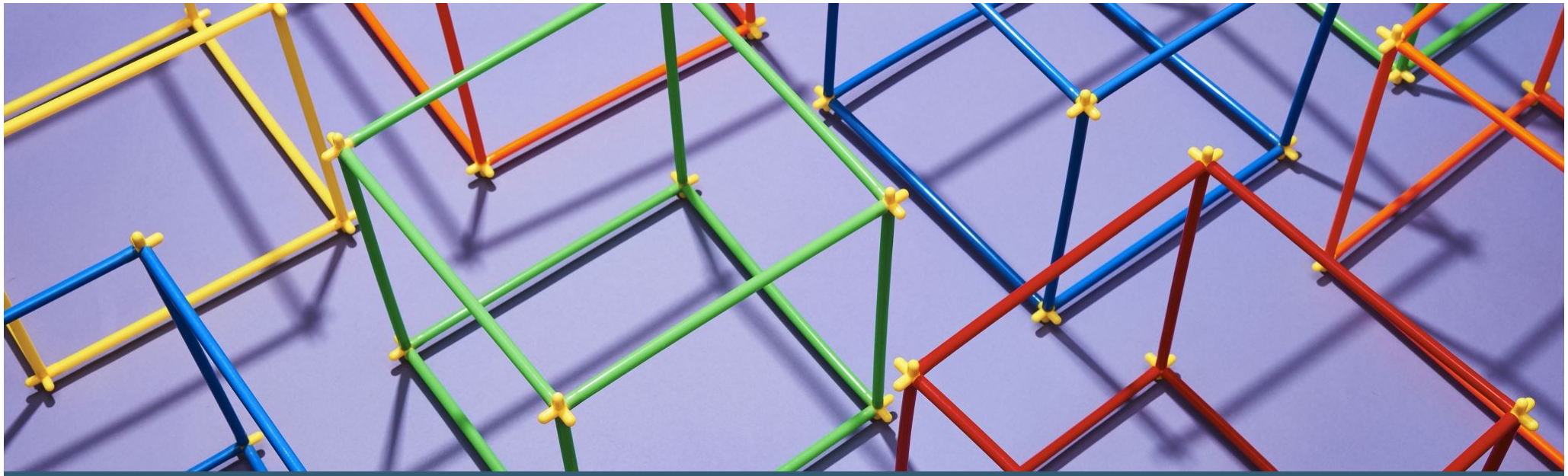
SQL Server

```
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

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

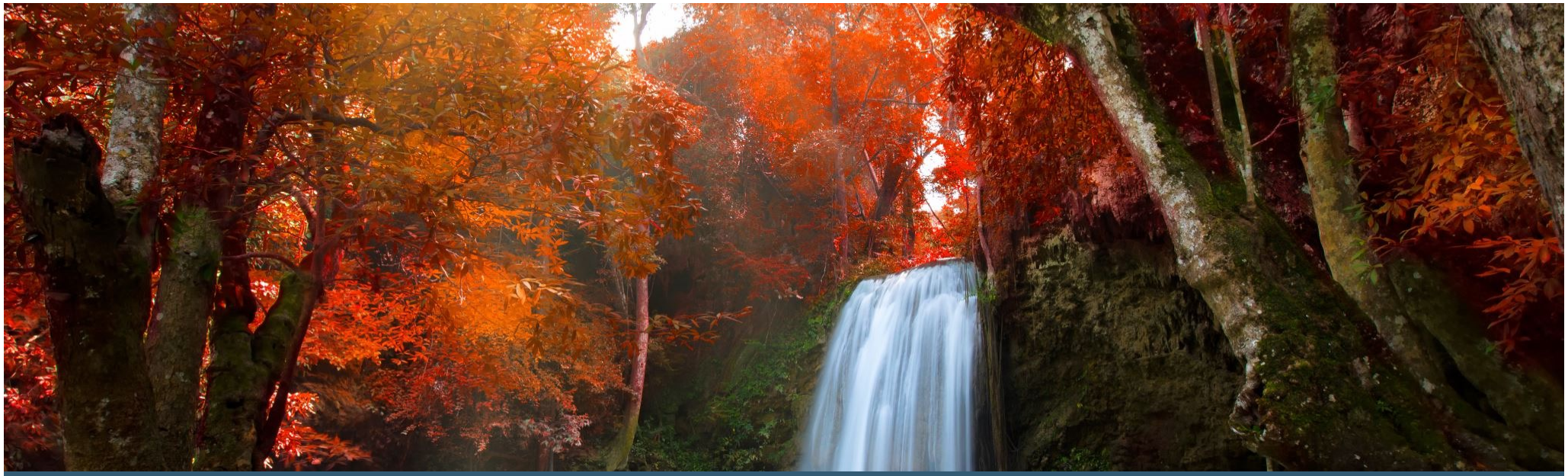
Stored Procedures

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

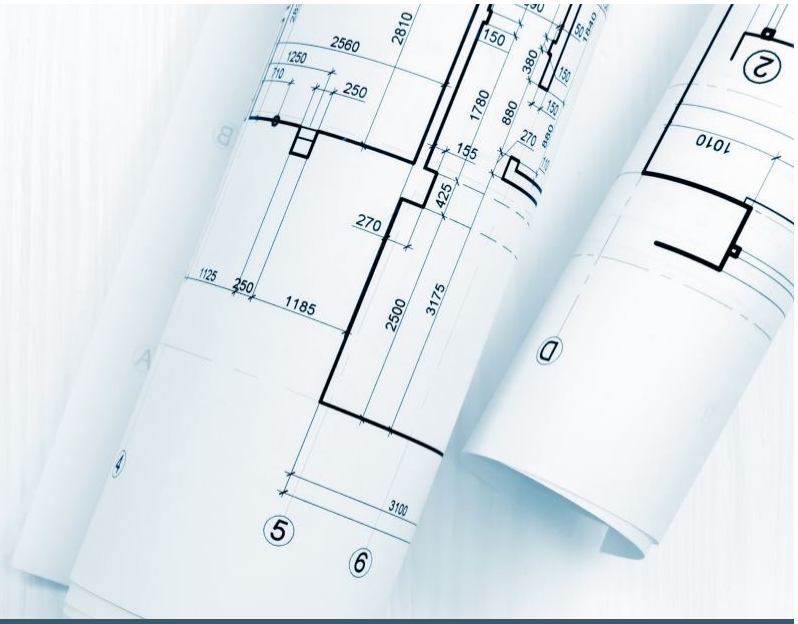
Postgres

```
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

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

```sql
SELECT count(*), DATE_PART('year',order_date)::int order_year
FROM sales.orders
WHERE customer_id=100
GROUP BY order_year
ORDER BY order_year DESC;
```

```
GroupAggregate  (cost=1824.35..1826.93 rows=102 width=12) (actual time=8.962..10.738 rows=4 loops=1)
  Group Key: ((date_part('year'::text, (order_date)::timestamp without time zone))::integer)
  Buffers: shared hit=900
  -> Sort  (cost=1824.35..1824.61 rows=107 width=4) (actual time=8.724..9.672 rows=107 loops=1)
        Sort Key: ((date_part('year'::text, (order_date)::timestamp without time zone))::integer) DESC
        Sort Method: quicksort  Memory: 30kB
        Buffers: shared hit=900
        -> Seq Scan on orders (cost=0.00..1820.74 rows=107 width=4) (actual time=0.0..7.5 rows=107 loops=1)
              Filter: (customer_id = 100)
              Rows Removed by Filter: 73488
              Buffers: shared hit=900
Planning Time: 0.224 ms
Execution Time: 10.894 ms
```

```sql
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
```

Community & Help

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?

Thank you!