SQL Server to PostgreSQL

Modernising and migrating workloads to Flexible Server for PostgreSQL in Azure.


Thank you sponsors!
INSPARK
twintos
ilionx
InfoSupport
Solid Innovator
redgate
sopra steria
DATA GIANTS
BRANDER
company
SQL
DATA
CRM
ERP
brander.nl
Bull
Master your AI destiny
dbWatch
BBTG
Data Saturdays

Speaker info

John Q. Martin

Global Partner Solutions Architect

Over fifteen years as a data professional working with SQL Server, PostgreSQL, Oracle, Snowflake, Databricks, and Redshift.

Previously worked for both AWS, Microsoft, and Sentry One, as well as working in the financial services sector.

John.q.martin@red-gate.com

/in/johnqmartin/



What are we covering

- Quick overview of Azure Database for PostgreSQL flexible server
- Assessing and figuring out how to take a SQL Server database to PostgreSQL.
- High level application considerations
  - Making the code changes
  - Moving the data
- Operating a PostgreSQL database, things you need to know

What is PostgreSQL flexible server?

- A fully managed PostgreSQL database engine hosted in Microsoft Azure.
  - Managed Linux VM with PostgreSQL installed, leveraging managed disks
- Supports native High Availability within an AZ or across AZs for increased resilience.
  - Warm standby that can be failed over to
- Has built-in support for PgBouncer to help with connection pooling and failover.

Built-in PgBouncer support
PgBouncer is a lightweight connection pooler that helps manage large numbers of client connections efficiently.
Explain why this matters: PostgreSQL is sensitive to connection counts, so PgBouncer stabilizes workloads under bursty or highly concurrent conditions.
Azure integrates PgBouncer so you don't need to deploy or manage it separately.

PgBouncer helps with connection pooling and enabling failover without connection string changes
Connection pooling reduces overhead and can dramatically improve throughput.
With PgBouncer in place, applications won't need to update connection strings during failovers—clients simply reconnect through PgBouncer and are routed to the active primary.
This simplifies application architecture and improves resilience.

Warm standby that can be failed over to
Clarify the "warm standby" concept: it's already running and receiving WAL logs, so failover times are fast.
Emphasize that this is automated—no manual intervention is required.

Managed Linux VM with PostgreSQL installed, leveraging managed disks
Behind the scenes, Azure provisions a Linux-based virtual machine dedicated to running PostgreSQL for your server.
Managed disks provide reliable performance, built-in redundancy, and predictable throughput.
Highlight that customers don't manage the OS—Azure does—but they still get predictable, tuned PostgreSQL performance.

Supports native High Availability within an AZ or across AZs
Flexible Server allows you to choose:

- **Same Availability Zone HA** for low-latency failover, or
- **Zone-redundant HA** for higher resilience.

There's a warm standby replica that continuously receives changes. If the primary fails, Azure automatically promotes the standby.

How does PostgreSQL differ from SQL Server?

- PostgreSQL stores data differently to SQL Server, leveraging heaps by default with indexes pointing to the data.
- Implementation differences
  - There are multiple index types available depending on the type of query activity you need to service.
  - Data types are similar, but the names and implementations differ slightly.
  - Procedures and functions are different in PostgreSQL to SQL Server.
- Configuration is important, parameter tuning for the instance plays a prominent role alongside query, index, and schema tuning.

Slide 7 – How does PostgreSQL differ from SQL Server?
• Storage model: PostgreSQL stores table data in heap files; indexes point to row locations. Expect different I/O patterns and the need for routine vacuuming to clean up dead tuples.
• Indexing strategy: Choose from multiple index types (B-tree, GIN, GiST, BRIN, etc.) depending on workload—not just B-trees as the default.
• Data types: Similar concepts but different names/behaviours (e.g., UUID, JSONB). Validate mappings in the data access layer.
• Configuration matters: Instance parameters (work_mem, shared_buffers, autovacuum settings) materially influence performance alongside schema and query design.
• Procedural code: Functions/procedures differ from T-SQL. Plan for rewrites in PL/pgSQL (or alternatives) and revisit error handling, transactions, and temp objects.

Assessing what needs to change

- Applications will need to be updated to support connectivity to the database.
- The data access layer in the application may need to be updated to account for different data types that need to be used.
- Think about other processes and systems which interact with your database currently.
  - Monitoring and Observability tooling
  - Backups
  - ETL/Analytics

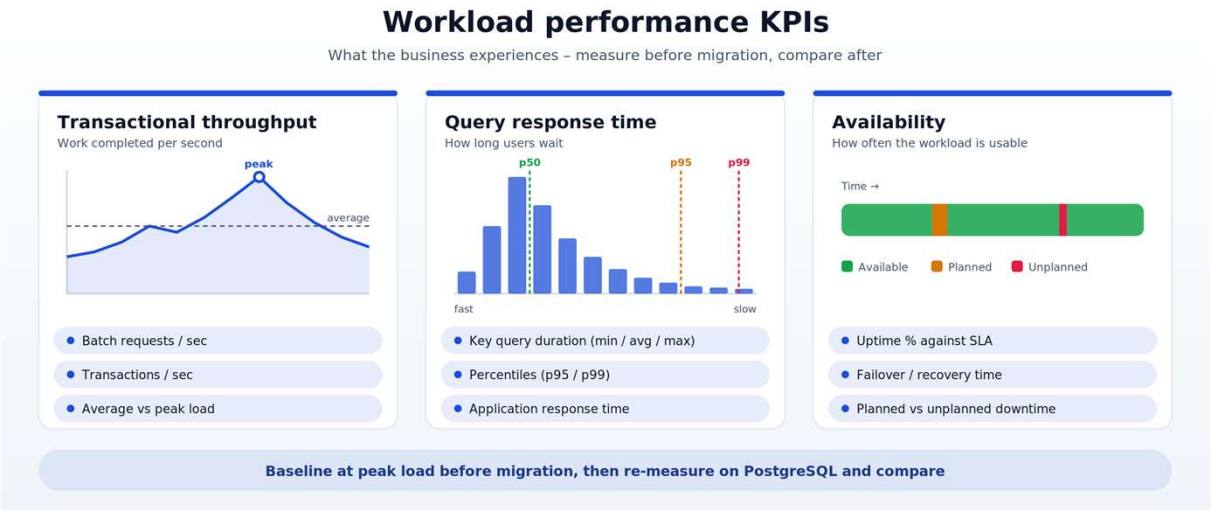
Slide 8 – Assessing what needs to change
• Application connectivity: Update drivers/ORMs and connection strings for PostgreSQL; confirm SSL/TLS and pooling behaviour.
• Data access layer: Reconcile data type differences and parameter placeholders; check for pagination, identity/sequence behaviour, and date/time handling.
• Surrounding systems: Inventory anything touching the DB—reporting, ETL, analytics, batch jobs, and integration services.
• Observability: Ensure monitoring, alerts, and logging pipelines understand PostgreSQL semantics.
• Backups & recovery: Align RPO/RTO with the new service's backup/restore model and test restores early.

Baselines are important

- When moving between different database engines it is vital that you understand the baseline levels of performance the current system works to.
  - What are key metrics to measure performance?
  - Resource usage will change.
  - Data storage will differ
- Think about how you will validate performance on the new database engine.
- Understand the price/performance ratio.

Slide 9 – Baselines are important
• Capture today's truth: Record throughput, latency, CPU, memory, storage IOPS/MBps, and key query timings before any change.
• Storage differences: Table/Index sizes and bloat characteristics will differ; plan headroom accordingly.
• Validation plan: Define what 'good' looks like on PostgreSQL (KPIs, SLAs) and how you will measure it after migration.
• Price/performance: Compare cost per transaction/query and total cost of ownership, not just raw SKU sizes.

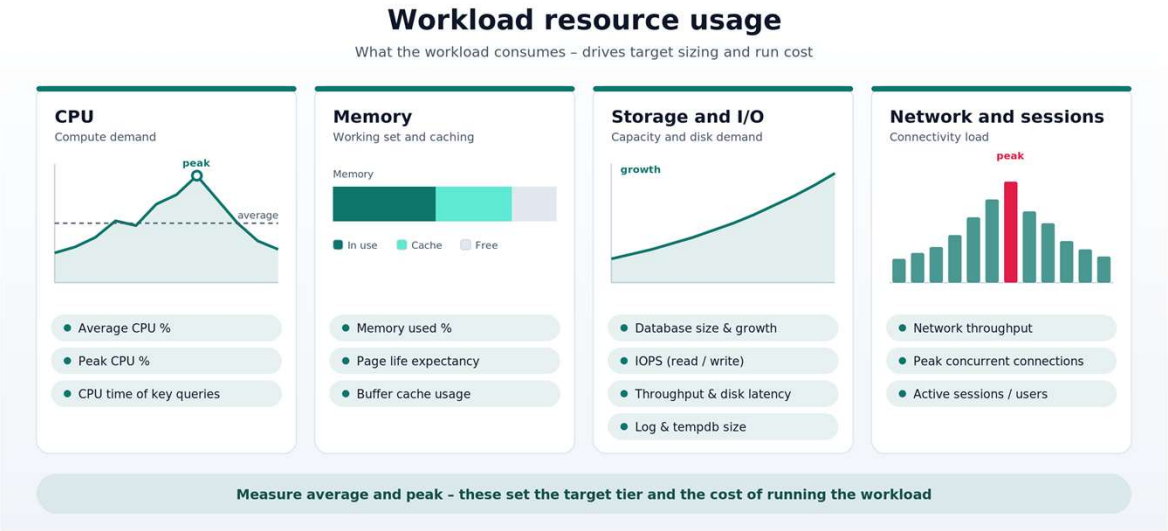
Baselines are important



Slide 9 – Baselines are important

- Capture today's truth: Record throughput, latency, CPU, memory, storage IOPS/MBps, and key query timings before any change.
- Storage differences: Table/Index sizes and bloat characteristics will differ; plan headroom accordingly.
- Validation plan: Define what 'good' looks like on PostgreSQL (KPIs, SLAs) and how you will measure it after migration.
- Price/performance: Compare cost per transaction/query and total cost of ownership, not just raw SKU sizes.

Baselines are important



Slide 9 – Baselines are important

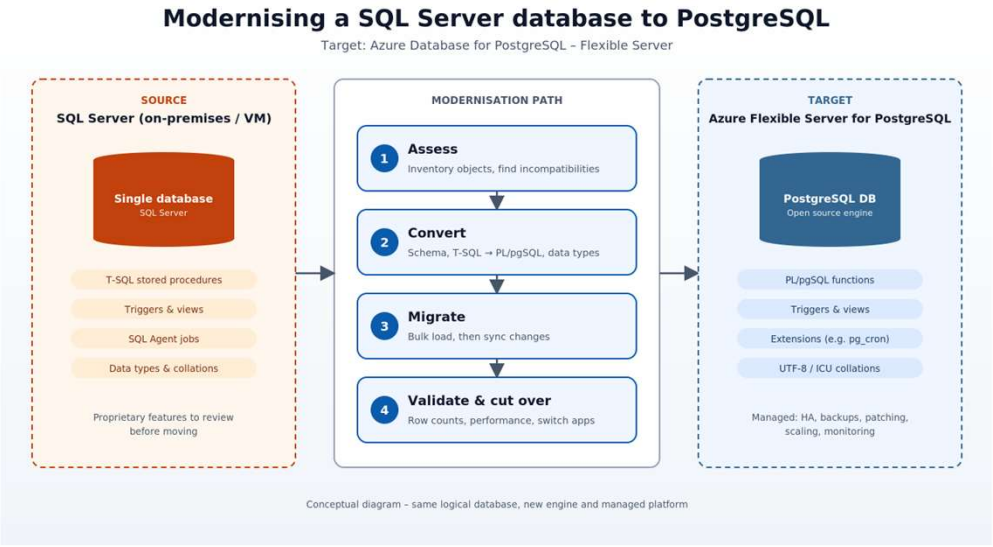
- Capture today's truth: Record throughput, latency, CPU, memory, storage IOPS/MBps, and key query timings before any change.
- Storage differences: Table/Index sizes and bloat characteristics will differ; plan headroom accordingly.
- Validation plan: Define what 'good' looks like on PostgreSQL (KPIs, SLAs) and how you will measure it after migration.
- Price/performance: Compare cost per transaction/query and total cost of ownership, not just raw SKU sizes.

Plan the change

- Think about what type of application/database are you moving.
  - Is it a monolith with many pieces resident in a single system, or are you moving a SOA/Microservices implementation
- Most successful modernisations/migrations take place in small pieces rather than big-bang.
- Think about breaking it down into stages.
  - Move non-programmatic elements first
  - Move the data
  - Finally move the procedures/functions/triggers/etc.

Slide 10 – Plan the change
• Classify the workload: Monolith vs. microservices influences how you slice the migration.
• Prefer incremental moves: Reduce risk by migrating in stages rather than a big-bang cutover.
• Order of operations: Move non-programmable objects first (schemas/tables), then data, then procedural code (functions/triggers).
• Change management: Feature-flag risky paths and schedule rehearsals in lower environments.

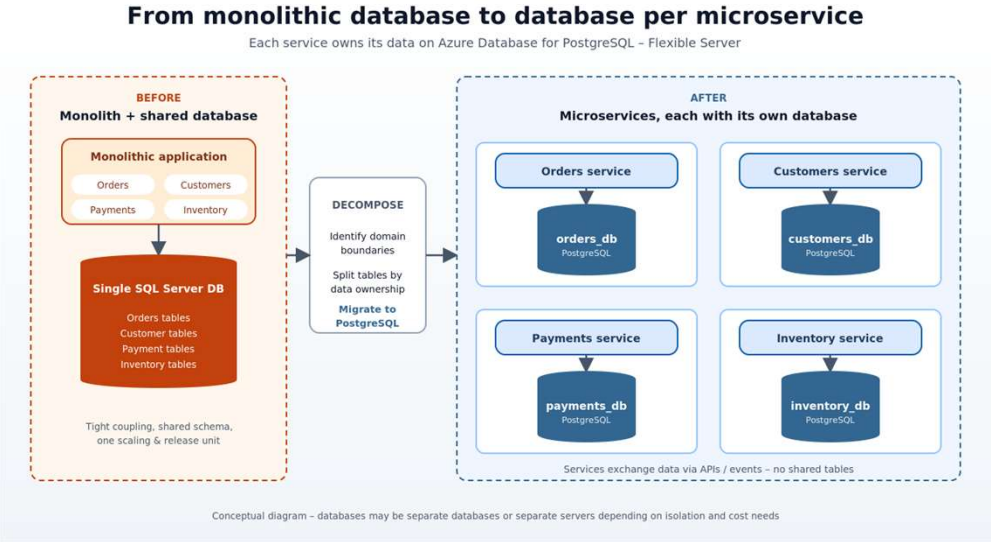
Plan the change



Slide 10 – Plan the change

- Classify the workload: Monolith vs. microservices influences how you slice the migration.
- Prefer incremental moves: Reduce risk by migrating in stages rather than a big-bang cutover.
- Order of operations: Move non-programmable objects first (schemas/tables), then data, then procedural code (functions/triggers).
- Change management: Feature-flag risky paths and schedule rehearsals in lower environments.

Plan the change



Slide 10 – Plan the change

- Classify the workload: Monolith vs. microservices influences how you slice the migration.
- Prefer incremental moves: Reduce risk by migrating in stages rather than a big-bang cutover.
- Order of operations: Move non-programmable objects first (schemas/tables), then data, then procedural code (functions/triggers).
- Change management: Feature-flag risky paths and schedule rehearsals in lower environments.

Things to watch out for

- Behavioural changes that you need to account for in the code objects like procedures, functions, and triggers.
- You're going to need extensions, vanilla PostgreSQL is very unlikely to work out of the box for you.
  - Because Flexible Server is a PaaS implementation, you will be limited in which extensions you can install and configure.
- You will need to pick the right type of index to service the queries, PostgreSQL has multiple types to pick from.

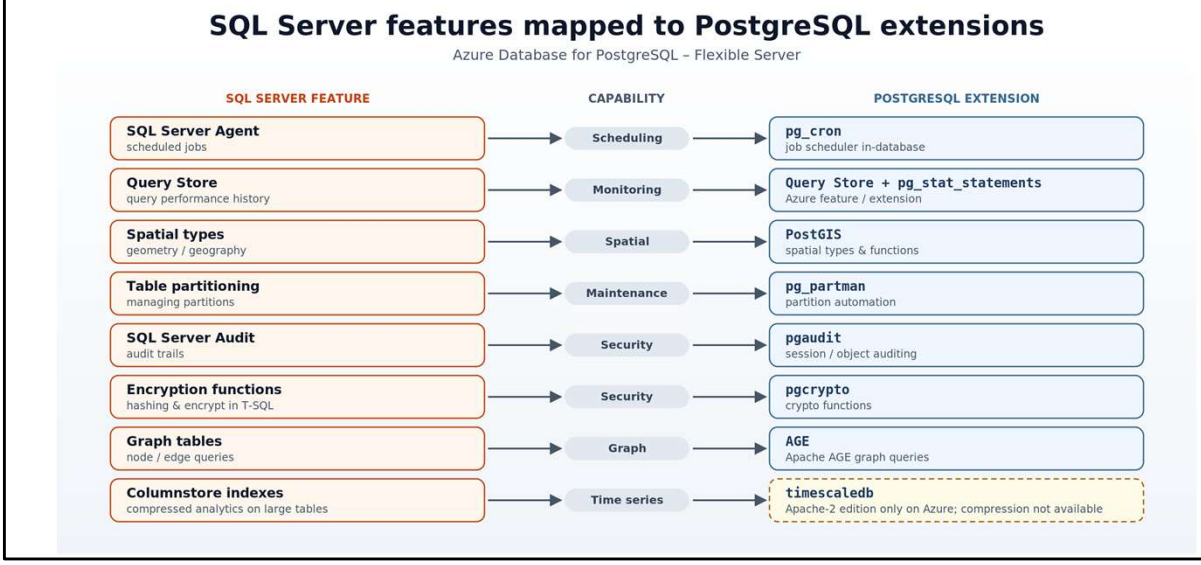
Slide 11 – Things to watch out for
• Behavioural differences: Transaction semantics, locking, and error handling can change outcomes—test critical code paths.
• Extensions are essential: Vanilla PostgreSQL rarely matches SQL Server features one-for-one; plan which extensions you need early.
• PaaS constraints: On Azure Flexible Server you can't install everything—confirm extension availability and version compatibility.
• Index choices: Pick the right index type per query pattern (equality, range, full-text, geospatial, time-series).

PostgreSQL extensions

- UUID-OSSP
  - Used for generating universally unique identifiers, are you using GUIDs in your SQL Server database and how are you generating them, you may need this.
- POSTGIS Extensions
  - If you are using geography types, then you will need one or more of these extensions.
- FDW extensions
  - Foreign data wrappers which allow you to query other databases including SQL Server, Oracle and PostgreSQL

Slide 12 – PostgreSQL extensions
• uuid-ossp: Generate UUIDs when your app or schema relies on GUID-like identifiers.
• PostGIS: Required for geospatial types and functions—enables spatial indexes and queries.
• FDWs: Foreign Data Wrappers allow querying external sources (including SQL Server/Oracle/PostgreSQL) for hybrid or phased migrations.
• Governance: Track which extensions are enabled per environment and how they impact backup/restore and upgrades.

PostgreSQL extensions



Slide 12 – PostgreSQL extensions

• uuid-ossp: Generate UUIDs when your app or schema relies on GUID-like identifiers.

• PostGIS: Required for geospatial types and functions—enables spatial indexes and queries.

• FDWs: Foreign Data Wrappers allow querying external sources (including SQL Server/Oracle/PostgreSQL) for hybrid or phased migrations.

• Governance: Track which extensions are enabled per environment and how they impact backup/restore and upgrades.

PostgreSQL indexes

- Data is stored in Heaps for PostgreSQL which will be scanned to retrieve data.
- There are multiple index types that can be used for different query workloads.



Slide 13 – PostgreSQL indexes
• Heaps + indexes: Tables are heaps; indexes accelerate access. Expect sequential scans when appropriate—don't fear them if cost-based planner prefers.
• B-tree: General-purpose for equality and range on ordered data.
• GIN: Inverted index—great for arrays, JSONB containment, full-text search.
• GiST/SP-GiST: Spatial, nearest-neighbour, and custom operator classes.
• BRIN: Summaries for very large, naturally ordered tables (time/ID)—tiny indexes, big scans.
• Hash/Bloom: Niche cases; validate benefits vs. maintenance cost. Based on bloom filters, **Performance**: They can outperform B-tree indexes for queries involving multiple columns, especially when the leading column of a B-tree index is not constrained.

Code conversion options

- Manually parse and convert the code from T-SQL to the appropriate SQL for the PostgreSQL object you need.
  - LLMs can assist with this process and make it a lot less onerous than it used to be
- PGLoader can be used to transfer schema objects that hold data, not great for converting procedures, functions etc.
- SQL Lines is another option for a tool to convert between dialects, but can struggle with complex code.

Slide 14 – Code conversion options
• Manual conversion: Highest fidelity—use for complex procs/functions, set-based rewrites, and transactional logic.
• LLM assistance: Speeds up boilerplate conversion; still review for correctness and performance.
• PGLoader/SQLines: Useful for schema/data objects; less effective for procedural code—expect manual follow-up.
• Build a test harness: Unit and integration tests to prove functional parity throughout conversion.

Migrating data between schemas

- ETL migration from one database to the other.
  - One and done approach for data lift and shift but will require downtime to move data
  - Adopt an incremental approach to minimise downtime with smaller final pre-cut-over data movement
- Commercial tooling like Omni Loader which can handle more continuous data migrations.
- Explore the use of the Foreign Data Wrapper extension for TDS databases to pull into the database via direct queries.

Slide 15 – Migrating data between schemas
• One-off lift & shift: Fastest path but requires downtime—use for smaller or less critical datasets.
• Incremental loads: Reduce downtime by syncing deltas and doing a short final cutover.
• Continuous replication tools: Consider commercial options when near-zero downtime is required.
• FDW for bridging: Use TDS/other FDWs to pull across data as queries during transition—good for validation and staged moves.

Operating a PostgreSQL database

- Similar but different to SQL Server, there are no DMVs by default.
  - Will need extensions including pg_stat_statements, pgrowlocks, auto_explain, pageinspect, pgaudit, pg_buffercache, and pg_freespacemap to fil the gap
- Vacuum is a vital process to run in order to clean out unused tuples.
  - PostgreSQL uses MVCC so tuples are valid between different transaction numbers
  - If you do not vacuum enough, you can run out of transaction numbers and your workload stops

Slide 16 – Operating a PostgreSQL database
• Observability via extensions: Add pg_stat_statements, auto_explain, and others to fill DMV-like gaps.
• MVCC & vacuum: Dead tuples accumulate; tune autovacuum and run manual vacuum/analyse where needed to avoid bloat and xid wraparound.
• Capacity hygiene: Monitor table/index bloat, connection counts, and checkpoint/ WAL behaviour.
• Security & auditing: Consider pgaudit and role design; verify TLS, network rules, and least-privilege access.

Summary

- Understand what it is that you are moving and the scope of work that you will need to undertake.
- Break the migration down into smaller chunks wherever possible and move it piece by piece.
- Benchmark and baseline your existing system to know what you need to have the new system perform at.
- Understand the differences between PostgreSQL and SQL Server when it comes to operating the platform.

Slide 17 – Summary
• Know your scope: What you're moving, dependencies, and acceptance criteria.
• Baseline first: Capture current performance and cost so improvements/regressions are obvious.
• Migrate in pieces: Prioritise low-risk components; move schema/data before code objects.
• Embrace PostgreSQL: Plan for extensions, indexing strategies, and operational differences (vacuum/MVCC).
• Validate & iterate: Test, measure, and tune before, during, and after cutover.

Rate This session

Modernizing workloads from SQL Server to PostgreSQL with Flexible Server for PostgreSQL




Thank you sponsors!
INSPARK
twintos
ilionx
InfoSupport
Solid Innovator
redgate
sopra steria
DATA GIANTS
BRANDER
company
SQL
DATA
CRM
ERP
brander.nl
Bull
Master your AI destiny
dbWatch
BBTG
Data Saturdays

Speaker info



John Q. Martin

Global Partner Solutions Architect

Over fifteen years as a data professional working with SQL Server, PostgreSQL, Oracle, Snowflake, Databricks, and Redshift.

Previously worked for both AWS, Microsoft, and Sentry One, as well as working in the financial services sector.

John.q.martin@red-gate.com

/in/johnqmartin/