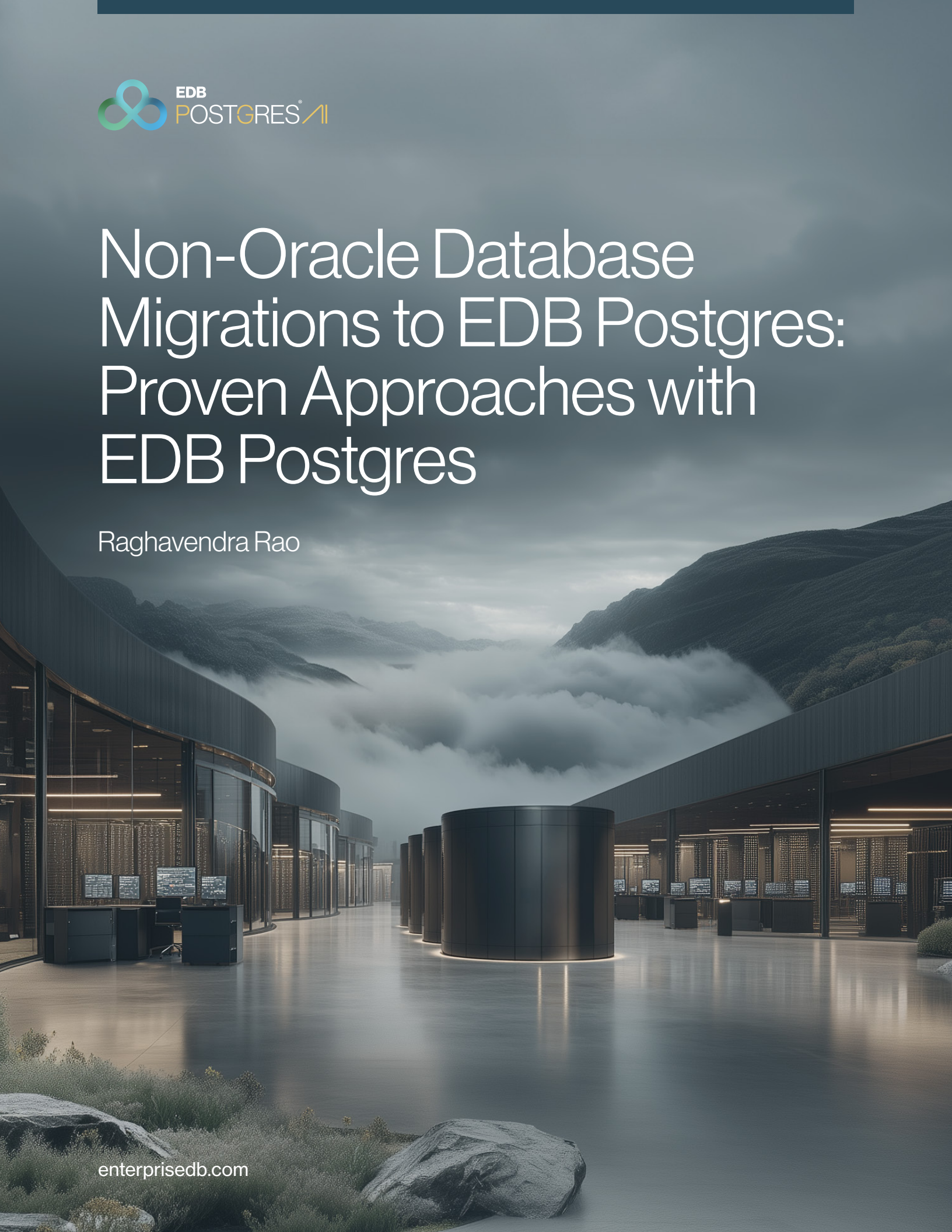
EDB POSTGRES AI

Non-Oracle Database Migrations to EDB Postgres: Proven Approaches with EDB Postgres

Raghavendra Rao



enterprisedb.com

Table of Contents

Objective 3

Introduction to non-Oracle migrations 3

Migration approach framework 3

Migration from MS SQL Server 4

- A - Assessment and architectural differences 4
- C - Conversion (schema and DDL) 5
- T - Tooling and planning 6
- I - Implementation (data and migration) 7
- O - Operation (monitoring, automation, observability) 7
- N - Normalization and validation 8
- S - Switch and support 9

Migration from CockroachDB 10

- A - Assessment and architectural differences 10
- C - Conversion (schema and DDL) 11
- T - Tooling and planning 11
- I - Implementation (data and migration) 12
- O - Operation (monitoring, automation, observability) 13
- N - Normalization and validation 14
- S - Switch and support 14

Migration from MongoDB 16

- A - Assessment and architectural differences 16
- C - Conversion (schema and DDL) 16
- T - Tooling and planning 17
- I - Implementation (data and migration) 18
- O - Operation (monitoring, automation, observability) 18
- N - Normalization and validation 19
- S - Switch and support 20

About the author 20

Objective

This best-practices guide provides a comprehensive, industry-standard architecture for migrating enterprise applications and data from **non-Oracle** databases—specifically **Microsoft SQL Server**, **CockroachDB**, and **MongoDB**—to **EDB Postgres**. The goal is to lead IT professionals and decision-makers through the whole migration lifecycle, from initial evaluation and planning to execution, validation, and production cutover.

The document outlines the most significant technological differences between the source and target platforms, discusses the common issues that arise during migration, and provides practical solutions and guidance on how to mitigate them. It uses **EDB-recommended tools** and processes to give practical assistance for streamlining migration operations; lowering operational and business risks; and ensuring optimum performance, security, and maintainability of EDB Postgres workloads. Organizations may support their long-term digital transformation goals by embracing these best practices and achieving a seamless, cost-effective, and future-ready move to EDB Postgres.

Introduction to non-Oracle migrations

- Brief overview of the migration journey: **Assessment**, **Planning**, **Migration**, **Validation**, and **Go-Live**
- Migration catalysts/scenarios
- Highlight the technical advantages of moving to EDB Postgres from MS SQL, CockroachDB, and MongoDB; highlight benefits per unique scenario

Migration approach framework

The **ACTIONS** framework will be used—a proven, seven-phase methodology for migrating from **Microsoft SQL Server**, **CockroachDB**, and **MongoDB** to **EDB Postgres**.

The framework provides a proven, repeatable process that guides IT professionals through the entire migration lifecycle, minimizing risks and ensuring successful outcomes. Each phase builds upon the previous one, creating a logical progression from understanding the current state to executing the migration and establishing ongoing operations.

The **ACTIONS** framework:

A - Assessment and architectural compatibility: Analyze the current state, evaluate compatibility differences, identify application challenges, and define migration requirements.

C - Conversion (schema and DDL): Convert database structures and definitions from source syntax to PostgreSQL-compatible formats.

T - Tooling and planning: Select migration tools, develop detailed plans, establish success criteria, and implement risk-mitigation strategies.

I - Implementation (data migration): Execute data migration, optimize performance, and handle data transformations.

O - Operation: Establish monitoring, automation, and observability for ongoing database health post-migration.

N - Normalization and validation: Conduct compliance checks, data validation, and testing to ensure accuracy and regulatory adherence.

S - Switch and support: Execute controlled cutover, provide post-migration monitoring, and complete operational handoff.

This framework applies consistently across SQL Server, CockroachDB, and MongoDB, while addressing each platform's unique architectural differences and migration challenges. Each database section follows this process with platform-specific guidance.

Migration from MS SQL Server

A - Assessment and architectural differences

Core architectural differences

- **Platform architecture:** SQL Server's **integrated Windows ecosystem** versus PostgreSQL's **multi-platform design**. SQL Server's **single-vendor stack** (SSMS, SSIS, SSRS) versus PostgreSQL's ecosystem of independent tools. **T-SQL procedural language** versus PL/pgSQL with different syntax and capabilities. **SQL Server Agent** jobs versus external schedulers (pg_cron, cron, Airflow).
- **Storage and performance:** SQL Server's **clustered index tables** versus PostgreSQL's **heap storage with separate indexes**. **Columnstore indexes** for analytics versus PostgreSQL's different approaches (BRIN, partitioning). **In-memory OLTP tables** versus PostgreSQL's standard buffer cache model. **Query optimizer** differences affecting execution plans and hints.
- **Transaction and consistency models:** SQL Server's **default READ COMMITTED with locking** versus PostgreSQL's **MVCC implementation**. **Snapshot isolation** differences in implementation and performance impact. **Distributed transactions** via MSDTC versus PostgreSQL's 2PC with different tooling. **Lock escalation** behavior versus PostgreSQL's row-level locking approach.
- **Data types and compatibility:** - DATETIME/DATETIME2 → TIMESTAMP/TIMESTAMPTZ (time zone handling differences) - NVARCHAR/NCHAR → VARCHAR/CHAR (PostgreSQL handles Unicode natively) - UNIQUEIDENTIFIER → UUID - MONEY → NUMERIC/DECIMAL - BIT → BOOLEAN - IMAGE/TEXT → BYTEA/TEXT (deprecated types)
- **Programming and procedural differences:** - T-SQL vs PL/pgSQL syntax variations - Error handling: TRY-CATCH vs EXCEPTION blocks - Temporary tables: #temp vs CREATE TEMP TABLE - Table variables (@table) have no direct equivalent - Dynamic SQL: sp_executesql vs EXECUTE
- **System architecture components:** - SQL Server Agent jobs → pg_cron or external schedulers - Service Broker → pg_message_queue or external solutions - Linked Servers → Foreign Data Wrappers (FDW) - Resource Governor → connection poolers with resource limits

Application challenges identification

- **Connection and driver changes:** Migration from ADO.NET SqlClient to Npgsql or EDB.NET Connector, including connection string format changes and authentication method updates (Windows Auth to certificate/LDAP)
- **T-SQL to PostgreSQL syntax:** Query refactoring for NOLOCK hints, TOP → LIMIT, ISNULL() → COALESCE(), string concatenation operators, and built-in function replacements
- **ORM and framework updates:** Entity Framework provider changes, Dapper query adjustments, and LINQ translation differences that may require code modifications
- **Case sensitivity impact:** SQL Server's case-insensitive default vs PostgreSQL's case-sensitive behavior affecting object names, string comparisons, and query results
- **Error handling patterns:** Different SQLSTATE codes, exception hierarchies, and retry logic requirements for deadlocks and timeouts
- **Performance optimization:** Query hint removal, parameter sniffing alternatives, and different approaches to query plan management

Assessment methods

- **AWS Schema Conversion Tool (SCT):** This stands as the most comprehensive free assessment tool, connecting directly to SQL Server instance to analyze schemas, stored procedures, and functions, then generating detailed PDF assessment reports that show what percentage can be automatically converted (typically 60%–80%) and effort estimates in hours for manual conversion work.
- **ora2pg:** Though primarily designed for Oracle, it can connect to SQL Server via FreeTDS/ODBC drivers to produce HTML assessment reports with migration difficulty scores from A (simple) to E (difficult) and person-day estimates, making it a viable free alternative despite requiring more setup effort.
- **Pre-flight check tools**
  - **pgloader --dry-run:** This tests what will convert without actually migrating and shows which tables and constraints will work and which will fail. Output can be saved as a text file to identify problem areas before migration.
  - **EDB Migration Toolkit:** It runs test migrations to identify conversion failures. The logs show exactly which stored procedures and functions won't convert and why. Use these errors to list items that need manual fixes.

C - Conversion (schema and DDL)

Schema conversion approach

- **Automated conversion tools: AWS SCT** converts 60%–80% of schemas automatically, with detailed reports showing manual work needed. **EDB Migration Toolkit** migrates schemas with built-in T-SQL compatibility for EDB Postgres Advanced Server. **pgLoader:** This handles basic schema discovery via FreeTDS but offers limited procedural code support. **Ispirer SQLWays** specializes in complex T-SQL pattern recognition and procedural code translation.
- **Manual conversion requirements: CLR assemblies** require complete rewrite in PL/pgSQL or PostgreSQL extensions. **Service Broker** queues need to be redesigned using LISTEN/NOTIFY or message queues. **Cross-database queries** require Foreign Data Wrappers or schema consolidation. **Extended stored procedures** have no equivalent and will need architectural redesign.

Key conversion challenges

- **Data type mappings: DATETIME** to TIMESTAMP with precision differences. **NVARCHAR** to VARCHAR requires encoding verification. **MONEY** to NUMERIC(19,4). **VARCHAR(MAX)** to TEXT with different storage behavior. **Computed columns** to generated columns (PostgreSQL 12+) or triggers.
- **Identity and sequences: IDENTITY columns** convert to SERIAL/IDENTITY with different sequence management. **SCOPE_IDENTITY()** patterns need RETURNING clause or lastval(). **RESEED operations** require manual sequence adjustments. **Identity insert** handling differs significantly.
- **SQL Server–specific features: Filtered indexes** become partial indexes with different optimizer rules. **Indexed views** convert to materialized views with manual refresh. **Table-valued functions** convert to set-returning functions. **Temporal tables** need triggers or extension-based solutions. **Columnstore indexes** have no direct equivalent.
- **Schema organization:** SQL Server **database becomes schema** in PostgreSQL. **Cross-database joins** need FDW or consolidation. **Three-part naming** (server.db.schema) requires refactoring. **Linked servers** convert to Foreign Data Wrappers.

Conversion tools and workflow

- **Recommended approach:** Use AWS SCT for assessment and initial conversion. Apply EDB MTK for enterprises needing T-SQL compatibility. Use pgLoader for simple table migrations. Deploy Ispirer for complex procedural code. Create custom scripts for patterns not handled by tools.
- **Conversion priority:** Convert in order: tables/constraints → views → functions → stored procedures → triggers. Test each layer before proceeding. Document T-SQL patterns that need application code changes.

T - Tooling and planning

Tool selection strategy

The choice of migration tool depends primarily on your database complexity. For **straightforward migrations** with mainly tables and views, **pgLoader** provides a reliable open source option. When dealing with **enterprise systems** containing T-SQL stored procedures and functions, **EDB Migration Toolkit** offers better compatibility features. **AWS environments** benefit from the integrated **SCT and DMS combination**, while **complex procedural logic** often requires commercial solutions such as **Ispirer SQLWays**.

Most successful migrations employ a **toolchain approach**: AWS SCT for initial assessment, followed by your chosen migration tool for execution, with custom scripts filling any remaining gaps.

Migration planning framework

- **Discovery and preparation phase:** Start by **inventorying all SQL Server components** including Agent jobs, SSIS packages, and linked servers. Establish **performance baselines** to measure migration success and **map your SQL Server security model** to PostgreSQL equivalents. Create **test environments** that mirror your production setup for validation before the actual migration.
- **Execution strategy:** Adopt a **wave-based migration approach**, moving from development to test, then to noncritical systems before touching production. Each wave needs **defined rollback procedures** and **maintenance windows** sized appropriately for your data volumes. **Application cutover planning** runs parallel to database migration preparation.
- **Validation framework:** Build comprehensive validation covering **schema completeness**, **data integrity** through row counts and checksums, **application functionality testing**, and **performance benchmarking** against your baselines. Include **disaster recovery testing** to ensure that your new PostgreSQL environment meets business continuity requirements.

Critical planning considerations

Experience shows that **procedural code conversion** typically takes **two to three times initial estimates**. Test with **production-scale data volumes** to uncover performance issues early. Success depends on having **application teams ready** for code changes, **SQL Agent job replacements** identified, and staff **trained on PostgreSQL** operational differences.

I - Implementation (data and migration)

Migration strategy selection

- **Big-bang migration:** Complete database cutover during a maintenance window using **EDB Migration Toolkit or pgLoader**. Best for smaller databases (<100GB) with acceptable downtime windows. Simpler to execute but requires longer downtime.
- **Trickle migration (CDC approach):** Use **EDB Replication Server** for continuous synchronization between SQL Server and PostgreSQL, maintaining both systems in parallel until cutover. Enables near-zero downtime for large databases. Requires more complex setup but minimizes business disruption.
- **Hybrid approach:** Combine pgLoader for initial bulk load with CDC tools for incremental changes during testing phase. Balances complexity with downtime requirements. Most practical for medium to large databases.

Implementation tools

- **EDB Replication Server** provides enterprise CDC with trigger-based or log-based replication, supporting bidirectional sync for phased migrations. Ideal when downtime must be minimal.
- **EDB Migration Toolkit** executes offline migrations with integrated schema and data transfer, offering parallel processing and progress monitoring. Included with EDB Postgres Advanced Server subscriptions.
- pgLoader delivers high-performance open source data loading with automatic retry mechanisms and error handling. Excellent for initial loads with simple transformations using straightforward syntax: `pgloader mssql://source pgsql://target`

Performance optimization techniques

- **Pre-migration tuning:** Temporarily disables the foreign key constraints and triggers during bulk loads. Increase PostgreSQL memory parameters: **maintenance_work_mem** to 1-2GB, **checkpoint_segments** to 100+, and **wal_buffers** to 16MB.
- **During migration:** Execute parallel loads across multiple tables simultaneously. Use PostgreSQL **COPY** command for raw data insertion when possible. Split large tables into chunks for manageable processing.
- **Post-migration:** Reenable all constraints and triggers. Rebuild indexes with CONCURRENTLY option. Run ANALYZE on all tables for query optimizer statistics. Validate data integrity before cutover.

O - Operation (monitoring, automation, observability)

Monitoring essentials

- **Performance monitoring:** Enable **pg_stat_statements** extension for query performance tracking, replacing SQL Server's Query Store functionality. Monitor connection pools through **PgBouncer** or **Pgpool-II** statistics dashboards. Establish baseline metrics during migration to compare PostgreSQL performance against SQL Server historical data.
- **Key metrics to track:** Focus on **query response times**, **connection counts**, **replication lag**, and **transaction rates**. Configure PostgreSQL logging with **log_min_duration_statement** to capture slow queries. Monitor disk I/O and memory usage patterns that differ from SQL Server behavior.

Automation framework

- **Job scheduling:** Replace SQL Server Agent with **pg_cron** for database tasks or **external schedulers** (Airflow, cron) for complex workflows. Convert maintenance plans to PostgreSQL scripts scheduled through your chosen automation tool.
- **Backup automation:** Implement **pg_basebackup** for base backups with **WAL archiving** for point-in-time recovery. Consider **Barman** or **pgBackRest** for enterprise backup management with retention policies and automated recovery testing.
- **High availability:** Configure **streaming replication** with automated failover using **Patroni**. This replaces SQL Server Always On Availability Groups with PostgreSQL-native solutions.

Observability platform

- **Metrics and alerting:** Deploy **Prometheus with Grafana** for real-time metrics visualization, or use **native cloud monitoring** (CloudWatch, Azure Monitor) for managed instances. Set alerts for **connection saturation** (>80% of max_connections), **replication lag** (>10 seconds), and **disk space** (<20% free).
- **Application integration:** Implement **application performance monitoring** (APM) tools that understand PostgreSQL. Track end-to-end transaction flows from application to database. Compare against SQL Server SLAs to ensure that migration success meets business requirements.

N - Normalization and validation

Data validation strategy

- **Row count verification:** Compare table row counts between SQL Server and PostgreSQL as the primary validation metric. For SQL Server, use `COUNT(*)` from system tables. For PostgreSQL, leverage `pg_stat_user_tables.n_live_tup` for quick counts. Document any expected differences (system tables, excluded objects).
- **Data integrity checks:** Beyond counts, validate **sample data** from critical tables using checksums or hash comparisons. Focus on **financial data**, **date columns** (time zone handling), and **numeric precision** that may differ between platforms. For large tables, validate representative samples rather than full datasets.

Schema validation

- **Object completeness:** Verify that all database objects migrated successfully: tables, views, functions, procedures, and triggers. Compare column counts, data types, and nullable settings between source and target.
- **Constraint and index validation:** Confirm that **primary and foreign keys** are active and enforced. Validate that **check constraints** syntax translated correctly. Verify that indexes exist and match SQL Server covering indexes where applicable. Check that **sequence current values** align with source IDENTITY columns.

Compliance and sign-off

- **Critical checkpoints:** Ensure that **security permissions** match business requirements, not necessarily SQL Server's model. Verify **audit trail continuity** if required for compliance. Confirm that **encryption at rest and in transit** meets regulatory standards.
- **Acceptance criteria:** Define clear success metrics: 100% schema objects migrated, >99.9% data accuracy, performance within 20% of baseline, all critical business functions tested. Create a formal sign-off checklist with stakeholder approval requirements before cutover.

S - Switch and support

Go-live preparation

Conduct full cutover rehearsal in staging environment with documented timings. Prepare a rollback plan and final data synchronization strategy.

Application cutover

- **Final changes:** Update connection strings and database drivers (SqlClient to Npgsql, JDBC updates). Fix any remaining SQL syntax differences discovered in testing, especially error handling code.
- **Cutover process:** Set SQL Server to read-only, run final data sync, validate data, switch application connections, monitor closely. Keep SQL Server available as an immediate rollback option for the first **48–72** hours.

Post-migration support

- **Stabilization monitoring:** Track application error logs, query performance against baselines, and connection pool health. Focus on catching PostgreSQL-specific issues such as connection exhaustion or transaction isolation differences.
- **Support transition:** Train operations team on PostgreSQL-specific tasks: backup/restore, performance tuning, troubleshooting. Document key operational differences from SQL Server. Define escalation paths for database versus application issues.

Success handoff

- **Acceptance criteria:** Zero critical errors, performance within 20% of baseline, all business processes functional. Get formal sign-off before decommissioning SQL Server.
- **Knowledge transfer:** Deliver PostgreSQL runbooks, performance troubleshooting guide, and validated DR procedures. Schedule 30-day and 90-day follow-ups for emerging issues.

Migration from CockroachDB

A - Assessment and architectural differences

Core architectural differences

- **Distributed vs. centralized design:** CockroachDB uses **range-based data distribution** with automatic sharding across nodes, while PostgreSQL employs **table-based storage** on single nodes or primary/standby configurations. CockroachDB's **built-in geo-partitioning** requires manual implementation in PostgreSQL using declarative partitioning or Citus extension. Moving from CockroachDB's automatic load balancing to PostgreSQL requires external tools such as **HAProxy** or **PgBouncer**.
- **Transaction models:** CockroachDB defaults to **serializable isolation** with distributed transactions using two-phase commit (2PC), while PostgreSQL defaults to **read committed** with local transactions only. CockroachDB requires **NTP clock synchronization** for distributed consistency, whereas PostgreSQL relies on local system time. Applications need modification to handle PostgreSQL's different isolation levels and lack of automatic retry logic.
- **SQL and feature compatibility/missing CockroachDB features:** CockroachDB lacks **stored procedures**, **triggers**, and **custom data types** that PostgreSQL supports extensively. Limited **window function** support and **no data-modifying CTEs** (INSERT/Update/DELETE with RETURNING) require query rewrites. CockroachDB's **connection multiplexing** eliminates pooler needs, but PostgreSQL requires **PgBouncer** or **Pgpool-II** for connection management.
- **Data type considerations:** CockroachDB recommends **UUID primary keys** for distribution, while PostgreSQL traditionally uses **SERIAL/BIGSERIAL**. Both support **JSONB** but with different indexing strategies—CockroachDB's inverted indexes versus PostgreSQL's GIN/GiST. PostgreSQL offers **arrays**, **full-text search**, and **custom types** absent in CockroachDB, potentially simplifying some data models.

Application impact assessment

- **Connection and retry logic:** Applications built for CockroachDB include **automatic retry logic** for serialization errors (40001). PostgreSQL requires different retry patterns for deadlocks (40P01) versus serialization failures. Remove CockroachDB-specific **retry annotations** (@Retryable) and implement PostgreSQL-appropriate error handling.
- **Query pattern changes:** Applications using **follower reads** for read scaling need refactoring to use PostgreSQL read replicas with different connection strings. **AS OF SYSTEM TIME** queries require alternative implementations using triggers and history tables or temporal tables extension. Geographic queries assuming data locality need redesign for PostgreSQL's centralized model.
- **Driver and ORM modifications:** Update connection strings from CockroachDB cluster endpoints to PostgreSQL single-host or load-balanced endpoints. Modify ORMs expecting CockroachDB's **INSERT ... ON CONFLICT** behavior differences. Remove assumptions about **automatic UUID generation** and implement PostgreSQL's uuid-ossp or gen_random_uuid().

Assessment methods

- **Schema and feature analysis:** Export schemas using `cockroach dump --dump-mode=schema` and analyze manually, since no automated tools exist for this migration path. Query **crdb_internal** system tables to identify distributed features such as zone configurations and interleaved tables. Document **changefeed** usage that needs replacement with PostgreSQL logical replication or CDC tools.
- **Performance and workload assessment:** Baseline CockroachDB metrics, including distributed query performance and transaction retry rates. Identify queries leveraging **follower reads** and **AS OF SYSTEM TIME** that need PostgreSQL alternatives. Evaluate whether single-node PostgreSQL can meet throughput requirements without automatic sharding, considering read replicas or Citus if needed.

C - Conversion (schema and DDL)

Schema conversion approach

- **Direct PostgreSQL compatibility:** Since CockroachDB uses PostgreSQL wire protocol, basic tables, indexes, and simple constraints transfer directly. Use `pg_dump` from CockroachDB and `pg_restore` to PostgreSQL for compatible objects. This handles roughly **70%–80% of basic schema** without modification.
- **Manual conversion requirements: INTERLEAVE IN PARENT** tables must be converted to standard foreign key relationships, losing the co-location benefits. **Hash-sharded indexes** need redesigning as standard B-tree indexes with potential application-level sharding. **Zone configurations** and geo-partitioning require PostgreSQL partitioning with manual partition placement. **Computed columns** convert to generated columns (PostgreSQL 12+) or trigger-based solutions.

Key conversion challenges

- **Primary key and distribution:** CockroachDB's **UUID primary keys** optimized for distribution may cause performance issues in PostgreSQL's B-tree indexes. Consider switching to SERIAL/BIGSERIAL or implementing UUID v7 for better locality. **Hash-sharded primary keys** have no direct equivalent, requiring application-level sharding logic or Citus extension.
- **Index strategy changes: STORING** clauses in indexes must convert to PostgreSQL's INCLUDE syntax, but performance characteristics differ. **Inverted JSONB indexes** become GIN indexes with different query optimization patterns. **Distributed index scans** that were efficient in CockroachDB may require query rewrites for PostgreSQL's single-node execution.
- **Lost distributed features: INTERLEAVE IN PARENT** for co-location has no PostgreSQL equivalent, potentially impacting the join performance. **Zone configurations** for data placement require manual partitioning and tablespace management. **Follower reads** and **AS OF SYSTEM TIME** need application-level implementation using read replicas and temporal tables.

Conversion tools and workflow

- **Semiautomated approach:** Use `pg_dump --schema-only` for initial export, expecting **20%–30% failure rate** on import. Create **sed/awk scripts** to bulk-convert known patterns: INTERLEAVE removal, STORING to INCLUDE, computed column syntax. No dedicated tools exist for CockroachDB-specific conversions.
- **Conversion steps:** Export schema using `cockroach dump` or `pg_dump`. Run test import to identify all incompatibilities. Fix systematically: Remove INTERLEAVE relationships first, convert indexes, and then handle computed columns. Validate by comparing object counts and running sample queries from application.

T - Tooling and planning

Tool selection strategy

- **Available migration tools:** No dedicated tools exist for CockroachDB to PostgreSQL migration. Use **cockroach dump** for accurate schema export, **pg_dump/pg_restore** for compatible objects, and **pgloader** for bulk data transfer. Create custom scripts for INTERLEAVE tables and computed columns conversion.
- **Recommended toolchain:** Schema: cockroach dump → sed/awk scripts → pg_restore. Data: pgloader with error handling → custom validation scripts. Plan **70% effort on application refactoring**, 30% on database migration.

Migration planning framework

- **Phase 1: Assessment (1–2 weeks):** Inventory CockroachDB-specific features usage. Size PostgreSQL infrastructure for total capacity. Identify application code requiring changes.
- **Phase 2: Preparation (2–4 weeks):** Set up PostgreSQL with PgBouncer and read replicas. Convert schemas and test with sample data. Refactor application retry logic and connection handling.
- **Phase 3: Execution (1–3 weeks):** Migrate in waves: noncritical → read-heavy → core workloads. Run parallel operations for 30-day rollback capability. Monitor single-node performance bottlenecks.

Critical planning considerations

- **Complexity multipliers:** Add 2x time for multiregion data, 3x for heavy INTERLEAVE usage, 2x for extensive changefeeds. Simple migrations take 2–3 weeks; complex ones need 2–3 months.
- **Risk mitigation:** Test with production data volumes early. Document features lost in migration. Prepare rollback procedures for each phase.
- **Decision criteria:** Proceed if accepting single-region limitations and single-node performance meets SLAs. Reconsider whether requiring active-active multiregion or automatic sharding.

I - Implementation (data and migration)

Migration strategy selection

- **Online migration (minimal downtime):** Use logical replication via CockroachDB changefeeds to stream changes to PostgreSQL. Requires custom change data capture handlers, since no direct CDC tools exist. It is best for large databases requiring near-zero downtime but adds significant complexity.
- **Offline migration (maintenance window):** Execute full cutover using pgloader during planned downtime. This is a simpler approach suitable for databases under 500GB with acceptable 2–6 hour maintenance windows, and it is the most reliable for ensuring data consistency.
- **Hybrid approach:** Initial bulk load with pgloader, then use changefeeds for incremental updates during the testing phase. This approach balances complexity with downtime requirements. It is recommended for most migrations as it provides validation time before cutover.

Implementation tools

- **pgloader configuration:** Connect using PostgreSQL protocol: `pgloader pgsql://user@cockroachdb:26257/db pgsql://user@postgres:5432/db`. Configure WITH options for batch size, worker count, and error handling. Set `on error resume next` for handling conversion issues.
- **CockroachDB changefeeds:** Create changefeeds for CDC: `CREATE CHANGEFEED FOR TABLE mytable INTO 'webhook-https://[...]'`. Requires custom webhook endpoint to translate changes to PostgreSQL. Alternative: Use Kafka sink with Debezium PostgreSQL connector.
- **Data validation tools:** Use **pg_comparator** for row-level comparison between systems. Create custom checksums using MD5 on sorted data for large tables. Monitor row counts continuously during migration.

Performance optimization techniques

- **Premigration tuning:** This disables PostgreSQL autovacuum during bulk loads. Increase `maintenance_work_mem` to 2GB and `checkpoint_segments` to 256. Drop indexes and foreign keys; recreate after data load. Set `synchronous_commit = off` temporarily.
- **During migration:** Use pgloader's `CONCURRENCY` setting with 4–8 workers, based on CPU cores. Split large tables into chunks using range queries on primary keys. Load independent tables in parallel using multiple pgloader processes.
- **Post-migration optimization:** Reenable constraints and create indexes with the `CONCURRENTLY` option. Run `ANALYZE` on all tables for statistics. Reset configuration parameters to production values. Validate query performance against CockroachDB baselines.

O - Operation (monitoring, automation, observability)

Monitoring essentials

- **Performance monitoring differences:** Replace CockroachDB's built-in distributed metrics with PostgreSQL-focused monitoring. Enable **pg_stat_statements** for query analysis, replacing CockroachDB Console's SQL insights. Monitor **single-node resource limits** (CPU, memory, disk I/O) more closely since you lose distributed load spreading. Track connection pool metrics through **PgBouncer** statistics.
- **Key metrics transition:** Shift focus from range metrics and leaseholder distribution to **buffer cache hit ratio**, **checkpoint frequency**, and **vacuum activity**. Monitor **replication lag** for read replicas replacing follower reads. Track **transaction ID wraparound**, which doesn't exist in CockroachDB.

Automation framework

- **Job migration strategy:** CockroachDB's built-in scheduled backups need replacement with **pg_cron** or external schedulers (cron, Airflow). Convert CockroachDB backup schedules to **pgBackRest** or **Barman** automated jobs. Implement **custom health checks** to replace CockroachDB's automatic node liveness detection.
- **Operational automation:** Set up **Patroni** for automatic failover, replacing CockroachDB's built-in resilience. Configure **automated vacuum and analyze** schedules based on table activity. Create scripts for **partition management**, replacing CockroachDB's automatic range splits.

Observability platform

- **Metrics collection:** Deploy **Prometheus with postgres_exporter** for metrics, replacing CockroachDB's Prometheus endpoint. Use **Grafana dashboards** designed for PostgreSQL instead of CockroachDB's pre-built ones. Configure **log aggregation** for PostgreSQL logs, which are more verbose than CockroachDB's structured logs.
- **Alerting adjustments:** Create alerts for **PostgreSQL-specific issues:** long-running transactions, table bloat, approaching transaction ID wraparound. Remove CockroachDB-specific alerts for range unavailability, clock skew, and node liveness. Set tighter thresholds for single-node resources, since there's no automatic redistribution.

N - Normalization and validation

Data validation strategy

- **Row count verification:** Compare table row counts between CockroachDB and PostgreSQL using system tables. For CockroachDB, query `crdb_internal.table_rows` for approximate counts, or use `SELECT COUNT(*)` for exact numbers. In PostgreSQL, use `pg_stat_user_tables.n_live_tup` for quick estimates or actual counts for precision. Document expected differences from in-flight transactions during migration.
- **Data integrity validation:** Create checksums on sorted data subsets, since full table checksums may differ due to internal storage differences. Focus on **business-critical columns** rather than system columns such as crdb_internal_mvcc_timestamp. Validate **JSONB data** carefully, as internal ordering might differ between systems. Compare sample rows for tables with complex data types.

Schema validation

- **Object completeness:** Verify that all tables migrated, but expect differences in system-generated names for constraints and indexes. Confirm that **foreign keys** are enforced (CockroachDB may have had them disabled for performance). Check that **computed columns are converted** correctly to generated columns or triggers. Validate that **zone configurations** didn't result in missing partitions.
- **Feature compatibility check:** Ensure that **UUID columns** maintained their values during migration. Verify that **timestamp precision** matches, as CockroachDB uses microseconds by default. Confirm that **sequence values** are set correctly for tables that switched from UUID to SERIAL. Check that **partial indexes** replacing CockroachDB's storing clauses work correctly.

Compliance and sign-off

- **Performance baseline comparison:** Document that single-node PostgreSQL latencies may differ from distributed CockroachDB patterns. Accept **different query plans** as normal due to different optimizers. Set realistic expectations that some queries may be faster, others slower.
- **Acceptance criteria:** Define success as 100% data migrated, critical business functions operational, and performance within acceptable SLAs (may differ from CockroachDB). Create formal sign-off checklists acknowledging that lost distributed features are acceptable. Include verification that backup/restore procedures work in the PostgreSQL environment.

S - Switch and support

Go-live preparation

- **Cutover rehearsal:** Execute full migration in staging with production data volumes. Test application behavior with PostgreSQL's different isolation levels and without retry logic. Document exact timings and validate that rollback procedures work.

Application cutover

- **Connection updates:** Replace CockroachDB cluster URLs (port 26257) with PostgreSQL endpoint (port 5432). Remove CockroachDB-specific parameters and retry decorators. Configure connection pooling for single-node architecture. Update error handling for PostgreSQL error codes.
- **Cutover execution:** Stop writes to CockroachDB, run final data sync, validate critical data, and switch application connections. Keep CockroachDB running for 7–14 days as a rollback option.

Post-migration support

- **Stabilization monitoring:** Watch for connection exhaustion without multiplexing, single-node resource spikes, vacuum/bloat issues, and different lock contention patterns. Focus on issues unique to moving from distributed to single-node architecture.
- **Knowledge transfer:** Train team on PostgreSQL vacuum vs. CockroachDB garbage collection, manual failover vs. automatic resilience, and traditional backups vs. distributed snapshots. Document operational differences in runbooks.

Success handoff

- **Acceptance criteria:** Stable operation for 14 days, performance within SLAs, all critical processes functional. Document new performance baselines and archive CockroachDB data before decommissioning.
- **Long-term planning:** Schedule PostgreSQL performance tuning training. Plan for major version upgrades and potential scaling strategies if approaching single-node limits.

Migration from MongoDB

A - Assessment and architectural differences

Core architectural differences

- **Document vs. relational model:** MongoDB's **schema-less documents** versus PostgreSQL's **structured tables** with defined schemas. **Nested documents** require normalization or JSONB storage. **Dynamic fields** need predefined columns or EAV patterns. **Embedded arrays** must become separate tables or JSONB.
- **Scalability and distribution:** MongoDB's **automatic sharding** versus PostgreSQL's manual partitioning or Citus extension. **Replica sets** with automatic failover versus PostgreSQL requiring Patroni/repmgr. **Read preferences** need explicit read-replica routing in PostgreSQL.
- **Transaction and consistency:** MongoDB's **document-level atomicity** versus PostgreSQL's full ACID. **Eventual consistency** options versus PostgreSQL's strong consistency. **Change streams** need logical replication or LISTEN/NOTIFY. **Aggregation pipelines** require SQL with CTEs and window functions.

Application impact assessment

- **Data access changes:** Replace MongoDB drivers with PostgreSQL equivalents. **BSON ObjectId** needs UUID or SERIAL replacement. **Dot notation** queries require JSONB operators or schema redesign. **Bulk operations** have different syntax and performance characteristics.
- **Query pattern modifications:** find() queries need SQL SELECT or JSONB conversion. **Aggregation pipelines** become complex SQL with JOINs/CTEs. **Geospatial queries** require PostGIS. **Text search** moves from $text to PostgreSQL full-text search.
- **Business logic adjustments:** Remove **replica set retry logic**. Modify **sharding key** distribution to partitioning. Replace **GridFS** with Large Objects or external storage. Update **change stream** listeners to triggers or logical replication.

Assessment methods

- **Schema discovery:** Use **MongoDB Compass** for schema analysis and field statistics. Run **variety.js** to identify all fields and data types. Export with `mongoexport --jsonArray` for structure analysis.
- **Workload analysis:** Profile with **db.setProfilingLevel()** for query patterns. Analyze **explain()** plans for index usage. Monitor **OpLog** for writing volumes and patterns.
- **Complexity evaluation:** Count collections with inconsistent schemas. Measure document nesting depth for JSONB vs. relational decisions. Assess aggregation pipeline complexity for SQL conversion effort. Calculate GridFS data volume for storage planning.

C - Conversion (schema and DDL)

Schema conversion approach

- **Document to relational mapping: Flat documents** map directly to table rows with columns for each field. **Nested objects** become separate tables with foreign keys or JSONB columns based on query patterns. **Arrays** convert to one-to-many relationships or PostgreSQL arrays for simple types. **Mixed-type fields** require JSONB or separate columns per type.
- **Schema design decisions:** Choose **normalized tables** for frequently queried, consistent structures. Use **JSONB** for variable schemas or deeply nested data. Apply a **hybrid approach** with core fields as columns, variable data in JSONB. Consider **table inheritance** for polymorphic MongoDB collections.

Key conversion challenges

- **Identifier management:** MongoDB **ObjectId** (12-byte) doesn't map directly to PostgreSQL types. Use **UUID** for distributed ID generation or **BIGSERIAL** for simpler cases. Maintain **_id mapping table** during migration for reference integrity. Handle **compound _id** fields as composite primary keys.
- **Data type conversions: ISODate** converts to TIMESTAMP WITH TIME ZONE. **NumberLong** maps to BIGINT, **NumberDecimal** to NUMERIC. **Binary data** moves from BSON to BYTEA. **Regular expressions** stored as strings need application-level handling. The **undefined/null** distinction is lost in PostgreSQL.
- **Index translations: Compound indexes** map directly, but with different optimizer behavior. **Multikey indexes** on arrays need GIN indexes or normalization. **2dsphere/2d** indexes require PostGIS geography/geometry types. **Text indexes** convert to GIN indexes with tsvector. **TTL indexes** need pg_cron with DELETE jobs.

Conversion tools and workflow

- **Available migration tools: mongo2pg** provides basic schema inference and data migration. **pgloader** supports MongoDB with automatic schema detection via BSON inspection. Custom scripts using mongoexport/COPY provide better control. **ETL tools** (Talend, Pentaho) are suitable for complex transformations.
- **Conversion strategy:** Analyze collections with MongoDB Compass for schema patterns. Generate DDL using automated tools or manual design based on patterns. Create PostgreSQL schemas with JSONB fallback for unmapped fields. Build indexes based on MongoDB query patterns. Validate that foreign key relationships match MongoDB references.

T - Tooling and planning

Tool selection strategy

- **Available tools:** Use **pgloader** with MongoDB support for automated schema inference and data migration; **mongo2pg** for basic collection-to-table conversions. **Custom scripts** using mongoexport/COPY work for complex transformations. **Commercial ETL** (FiveTran, Stitch) are good for managed migrations with CDC.
- **Selection criteria:** Use pgloader for consistent schemas, custom scripts for complex transformations or GridFS. Plan **60% effort on schema design**, 40% on data migration.

Migration planning framework

- **Phase 1: Discovery (1–2 weeks):** Profile schemas with Compass and variety.js. Identify normalization vs. JSONB storage needs. Map aggregation pipelines to SQL. Size PostgreSQL infrastructure.
- **Phase 2: Design (2–3 weeks):** Create a PostgreSQL schema with tables/JSONB mix. Design indexes from query patterns. Plan a GridFS migration strategy. Build transformation scripts.
- **Phase 3: Migration (1–4 weeks):** Migrate by collection priority. Run parallel for independent collections. Validate after each batch. Test application before cutover.

Critical planning considerations

- **Complexity multipliers:** Add 2x time for inconsistent schemas, 3x for heavy aggregations, 2x for sharded clusters. Simple migrations: 2–4 weeks. Complex migrations: 2–3 months.
- **Risk mitigation:** Test JSONB vs. normalized performance early. Validate document size limits (PostgreSQL 1GB vs MongoDB 16MB). Plan for missing features (auto-sharding, change streams).
- **Go/no-go criteria:** Proceed if the relational model is acceptable and single-node performance is sufficient. Reconsider whether the situation needs flexible schema evolution or automatic sharding.

I - Implementation (data and migration)

Migration strategy selection

- **Offline migration (full cutover):** Export with `mongodump` (BSON) or `mongoexport` (JSON/CSV) and import via PostgreSQL COPY or pgloader. This works best for databases under 100GB with an acceptable downtime window. It's the simplest approach, with guaranteed consistency.
- **Online migration (CDC approach):** Use **MongoDB change streams** (requires replica set/sharded cluster) with custom connector to PostgreSQL. Alternatively, use **Debezium MongoDB connector** with Kafka to PostgreSQL sink. Enables near-zero downtime for large databases.
- **Hybrid approach:** Initial bulk load with pgloader, then change streams or Debezium for incremental updates. Recommended for most migrations, balancing complexity with downtime. Provides a validation period before cutover.

Implementation tools

- **pgloader configuration (v3.6+):** Connect using `pgloader mongodb://user:pass@host:27017/db postgresql://user:pass@host:5432/db`. This supports automatic schema detection from MongoDB 4.0+. Configure `WITH batch rows = 10000, batch size = 500MB` for optimal performance. Use `MATERIALIZE VIEWS` for complex transformations.
- **mongodump/pg_restore method:** Export BSON: `mongodump --db=mydb --collection=mycoll`. Transform using Python with pymongo and psycopg2. Use PostgreSQL 14+ `COPY FROM JSON` for direct JSON import. Leverage `\copy` with `FORMAT csv` for preprocessed data.
- **Change streams with PostgreSQL:** 14+ MongoDB 4.0+ change streams with `fullDocument` option for complete documents. Use **logical replication** in PostgreSQL for downstream processing. Consider **pg_partman** for time-based partitioning of streamed data. Handle idempotency with `INSERT ... ON CONFLICT`.

Performance optimization techniques

- **Premigration tuning (PostgreSQL 15+):** Use `maintenance_work_mem = 4GB` on modern systems. Set `max_wal_size = 10GB` for bulk loads. Enable `wal_compression = on` to reduce I/O. Consider `shared_buffers = 25%` of RAM for a dedicated migration server.
- **During migration:** Use MongoDB $sample for test migrations with representative data. Leverage PostgreSQL 14+ COPY FREEZE for faster loads into new tables. Enable parallel workers: `max_parallel_maintenance_workers = 4`. Use **table partitioning** for collections greater than 100GB.
- **Post-migration optimization:** Use PostgreSQL 15+ **MERGE** command for upserts from change streams. Run `VACUUM ANALYZE` with parallel workers. Enable `pg_stat_statements` for query performance comparison. Consider **BRIN indexes** for time-series data from MongoDB.

O - Operation (monitoring, automation, observability)

Monitoring essentials

- **Performance monitoring shift:** Replace MongoDB's **db.currentOp()** and profiler with PostgreSQL's **pg_stat_activity** and **pg_stat_statements**. Monitor **JSONB query performance**, which differs from MongoDB document queries. Track **vacuum activity** and table bloat, concepts absent in MongoDB. Watch **connection pool** behavior, since MongoDB connection multiplexing is lost.
- **Key metrics changes:** Shift from MongoDB's **replication lag** and **oplog window** to PostgreSQL's **WAL lag** and **replication slots**. Monitor **TOAST table** usage for large JSONB documents replacing GridFS. Track **index bloat** and **sequential scans** on JSONB fields. Replace MongoDB lock percentage with PostgreSQL **lock wait** events.

Automation framework

- **Job migration:** MongoDB **scheduled tasks** (Atlas scheduled triggers) need **pg_cron** or external schedulers. Replace **TTL indexes** with pg_cron DELETE jobs or partitioning with retention. Convert MongoDB **aggregation pipelines** in cron to PostgreSQL **materialized view refreshes**. Implement **automatic VACUUM/ANALYZE** schedules for high-write tables.
- **Operational automation:** Replace MongoDB's **automatic sharding** with manual partition management using pg_partman. Configure **pgBackRest** or **Barman** replacing MongoDB's point-in-time recovery. Set up **Patroni** for automatic failover versus MongoDB's replica set elections. Create **JSONB validation** triggers, replacing MongoDB schema validation.

Observability platform

- **Metrics collection:** Deploy **postgres_exporter** with custom queries for JSONB metrics. Use **pg_stat_monitor** (Percona) for better query analysis than MongoDB's profiler. Configure **pgBadger** for log analysis, replacing MongoDB's log parsing. Monitor **autovacuum** effectiveness with no MongoDB equivalent.
- **Alerting adjustments:** Create alerts for **JSONB query slowness** and **GIN index bloat**. Monitor **transaction ID wraparound** risk, not applicable in MongoDB. Alert on **replication slot lag**, replacing oplog size monitoring. Track **connection exhaustion** without MongoDB's connection multiplexing. Set thresholds for **table bloat** percentage unique to PostgreSQL.

N - Normalization and validation

Data validation strategy

- **Document count verification:** Compare MongoDB's `db.collection.countDocuments()` with PostgreSQL's `SELECT COUNT(*)`. Account for filtered migrations in which not all documents were migrated. Validate that collections split into multiple PostgreSQL tables match total counts. Document excluded documents (e.g., those with null required fields).
- **Data integrity validation:** Verify that **ObjectId to UUID/SERIAL** mappings maintained referential integrity. Compare sample documents using MongoDB's `findOne()` with PostgreSQL JSONB queries. Validate that **nested arrays** properly converted to related tables or PostgreSQL arrays. Check that **date fields** preserved timezone information during migration.

Schema validation

- **Structure verification:** Confirm that all MongoDB **collections** mapped to PostgreSQL tables or JSONB columns. Verify that **indexes** translated appropriately: compound, multikey, text, and geospatial. Validate that **unique constraints** from MongoDB unique indexes are enforced. Check **foreign keys** created for previously embedded document relationships.
- **Data type accuracy:** Verify that **NumberDecimal** preserved precision in NUMERIC columns. Confirm that **Binary data** is correctly stored in BYTEA. Validate **null vs undefined** handling (PostgreSQL only has NULL). Check that **mixed-type fields** are properly stored in JSONB or separate columns.

Compliance and sign-off

- **Query performance validation:** Compare MongoDB **aggregation pipeline** execution times with SQL equivalents. Document that JSONB queries may differ from native MongoDB performance. Validate that **full-text search** results match between MongoDB $text and PostgreSQL FTS. Verify that **geospatial queries** return the same results in PostGIS.
- **Acceptance criteria:** Define success as 100% critical data migrated, referential integrity maintained, and application queries functioning. Document performance differences as new baselines (some queries faster, others slower). Verify that backup/restore procedures work with PostgreSQL's different approach. Get sign-off acknowledging that loss of MongoDB-specific features (auto-sharding, change streams) is acceptable.

S - Switch and support

Go-live preparation

- **Cutover rehearsal:** Execute full migration in staging with production data volume. Test JSONB queries versus MongoDB operations. Validate connection pooling without MongoDB's multiplexing. Document migration timings.

Application cutover

- **Connection updates:** Replace MongoDB URIs (mongodb://) with PostgreSQL (postgresql://). Switch from MongoDB drivers to psycopg2/node-postgres. Remove MongoDB-specific options (readPreference, writeConcern). Configure PgBouncer for connection pooling.
- **Cutover execution:** Stop MongoDB writes, run final sync, validate critical data, switch connections. Keep MongoDB running for a 7–14 day rollback window.

Post-migration support

- **Stabilization monitoring:** Watch for JSONB query performance issues, connection pool exhaustion, vacuum lag on heavy-update tables, and lock contention on JSONB documents. Focus on issues unique to PostgreSQL's different architecture.
- **Knowledge transfer:** Train team on JSONB operators versus MongoDB queries, vacuum/analyze concepts, and index maintenance differences. Document troubleshooting procedures for JSONB performance issues.

Success handoff

- **Acceptance criteria:** Stable operation for 14 days, all features working, performance within SLAs. Document new performance baselines and get stakeholder sign-off.
- **Long-term planning:** Schedule JSONB optimization training. Plan potential normalization of heavily queried JSONB. Consider partitioning for large tables. Document path to relational model if needed.

About the author

Raghavendra Rao
Senior Practice Leader, Global Migrations, EDB
Connect on LinkedIn

Raghavendra Rao is a senior practice leader in global migration and an accomplished database professional with more than two decades of experience across enterprise and open source platforms. A passionate community contributor, he shares his expertise through blogging, presentations, and training, drawing inspiration from the global PostgreSQL community.

EDB POSTGRES AI

EDB Postgres AI is the first open, enterprise-grade sovereign data and AI platform, with a secure, compliant, and fully scalable environment, on premises and across clouds. Supported by a global partner network, EDB Postgres AI unifies transactional, analytical, and AI workloads, enabling organizations to operationalize their data and LLMs where, when, and how they need them. For more information, visit enterprisedb.com

© EnterpriseDB Corporation 2026. All rights reserved.