Postgres Ankara 2026

Migrating to PostgreSQL with Zero Downtime

Mehmet Emin KARAKAŞ
Database Manager

Agenda

1. Migration architecture decisions
2. Compatibility deep-dive
3. Data movement at scale
4. Validation and performance proof
5. Zero-surprise cutover and operations

From MySQL to Postgres

- Move from engine-specific behavior to standards-driven SQL
- Replace fragile implicit conversions with explicit data contracts
- Improve indexing, planning, and diagnostic visibility
- Enable safer HA/DR and recovery workflows
- Build a scalable platform for long-term growth

Session scope (DBA deep-dive)

- Focus on OLTP migrations (InnoDB -> PostgreSQL)
- Service fleets with 1 GB to multi-TB datasets
- Downtime target: minutes, not hours
- No application rewrite assumption
- Goal: safe migration with measurable performance gains

Decision matrix: choose migration strategy

Strategy	Downtime	Complexity	Best for
Big bang	High	Low	Small monoliths, <200 GB
Phase by domain	Medium	Medium	Microservices with clear boundaries
CDC + dual run	Low	High	Critical systems, TB scale

- If RTO < 30 minutes, CDC is usually mandatory.

Reference migration architecture

```
MySQL binlog -> Debezium -> Kafka -> sink -> PostgreSQL
```

- Initial load: `pgloader` / batch `COPY`
- Change stream: row-based binlog events
- Reconciliation: checksum jobs + business assertions
- Controlled cutover: write freeze + final lag drain
- Rollback window: keep MySQL read-only and tailing

Pre-migration discovery checklist

- Capture schema, routines, triggers, events
- Profile table sizes, write rates, and growth
- Identify MySQL-only SQL modes and ORM assumptions
- Map SLO-critical endpoints to underlying queries
- Baseline p95/p99 latency and error budgets
- Lock migration scope and freeze risky DDL

Character set and collation traps

- MySQL defaults often use case-insensitive collations
- PostgreSQL text comparison is collation-dependent and often case-sensitive
- Unique index behavior can change for user-facing identifiers
- Decide early: `citext`, functional indexes, or normalized values

```
CREATE EXTENSION IF NOT EXISTS citext;
ALTER TABLE account
  ALTER COLUMN email TYPE citext;
```

Type mapping deep-dive

MySQL	PostgreSQL	Risk
`TINYINT(1)`	`boolean`	app expects `0/1` integer
`DATETIME`	`timestamptz`	timezone bugs
`JSON`	`jsonb`	operator/query rewrite
`ENUM`	enum/domain/table	deployment friction
`DOUBLE`	`double precision` / `numeric`	precision drift
`UNSIGNED BIGINT`	`numeric(20)` or `bigint` check	overflow edge cases

UUID mapping for MySQL -> PostgreSQL

- Common CDC issue: MySQL `BINARY(16)` UUIDs land as bytes, not `uuid`
- Use Kafka Connect SMT to convert binary <-> canonical UUID text
- Recommended repo: `emin100/uuid_smt`
- Supports both directions and nested Debezium payloads

https://github.com/emin100/uuid_smt

Auto-increment and sequence correctness

```sql
ALTER TABLE orders
  ALTER COLUMN id
  ADD GENERATED BY DEFAULT AS IDENTITY;

-- After bulk load:
SELECT setval(
  pg_get_serial_sequence('orders', 'id'),
  COALESCE((SELECT MAX(id) FROM orders), 1),
  true
);
```

SQL behavior differences that break silently

- `GROUP BY` : PostgreSQL enforces correctness more strictly
- `ONLY_FULL_GROUP_BY` assumptions differ by app history
- `NULL` ordering differs unless explicit `NULLS FIRST/LAST`
- `INSERT ... ON DUPLICATE KEY` -> `ON CONFLICT`
- Backticks, implicit casts, and permissive conversions disappear

Rewrite pattern: upsert

```sql
-- MySQL
INSERT INTO inventory (sku, qty)
VALUES ('A-100', 10)
ON DUPLICATE KEY UPDATE qty = qty + VALUES(qty);

-- PostgreSQL
INSERT INTO inventory (sku, qty)
VALUES ('A-100', 10)
ON CONFLICT (sku)
DO UPDATE SET qty = inventory.qty + EXCLUDED.qty;
```

Non pk tables

- PK : With upsert mode
- Non PK: With insert mode

CDC timezone settings

KAFKA_OPTS=-Duser.timezone=Europe/Istanbul

Query planning differences

- PostgreSQL does not use MySQL optimizer hints directly
- Replace hint-driven tuning with statistics + index design
- Enable `pg_stat_statements` before workload replay
- Use `EXPLAIN (ANALYZE, BUFFERS, WAL)` for evidence
- Rebuild expensive query paths with covering indexes carefully

Example tuning workflow

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT customer_id, SUM(amount)
FROM payment
WHERE created_at >= now() - interval '30 days'
GROUP BY customer_id
ORDER BY SUM(amount) DESC
LIMIT 50;
```

- Add multicolumn or partial indexes based on filter/selectivity
- Re-check plans after `ANALYZE` on loaded tables

Data movement patterns by size

- <20 GB: single-pass load + short freeze
- 20 GB - 2 TB: parallel chunk load + CDC catch-up
- > 2 TB: partition-first strategy + phased cutover
- Large objects/BLOBs: separate channel and checksum policy
- Always throttle load to protect source and network

`pgloader` baseline config

```
LOAD DATABASE
  FROM mysql://user:pass@mysql/app
  INTO postgresql://pg:pass@pg/app
WITH include drop, create tables, create indexes, reset sequences
SET work_mem to '256MB', maintenance_work_mem to '2GB'
SET synchronous_commit to 'off'
CAST type datetime to timestamptz using zero-dates-to-null;
```

- Tune memory knobs by host capacity, not by template.

CDC lag management during dual run

- Define acceptable lag per domain (e.g. <= 5 sec)
- Alert on lag slope, not only absolute value
- Backpressure sinks when PostgreSQL write saturation appears
- Keep idempotent consumers for replays
- Record cutover checkpoint (binlog position / GTID)

Debezium Lag Checker

- Monitors Kafka Connect / Debezium connector and task states via REST API
- Prometheus exporter for connector lag and status metrics
- Sends alerts to Microsoft Teams via webhook
- Supports multi-instance monitoring and Docker auto-discovery

https://github.com/emin100/debezium_lag_checker

Validation framework: 4 layers

1. **Structural:** schema parity, constraints, indexes
2. **Quantitative:** row counts and null distributions
3. **Digest:** checksums for deterministic slices
4. **Business:** financial/inventory/order reconciliations

If layer 4 fails, migration is not complete even if 1-3 pass.

Example reconciliation SQL

```sql
-- Per-day revenue comparison
SELECT order_date, SUM(total_amount) AS revenue
FROM orders
GROUP BY order_date
ORDER BY order_date;

-- Compare MySQL export vs PostgreSQL result in CI gate
```

Performance proof before go-live

- Replay top N production reads against PostgreSQL
- Compare p95/p99 and plan stability
- Test lock-heavy write paths with realistic concurrency
- Verify connection pooling limits (`pgBouncer` transaction mode)
- Burn-in for at least one business cycle

Cutover runbook (minute-by-minute)

1. T-N: start CDC connector from mysql to postgresql
2. T-N: load data with pgloader
3. T-N: start CDC sink mysql to postgresql
4. T-5: drain CDC lag to zero
5. T-3: run checksum quick pack + sequence sync
6. T-2: start cdc connector from postgresql to mysql
7. T-1: switch DNS/connection strings

Cutover runbook 2 (minute-by-minute)

8. T+0: enable writes on PostgreSQL
9. T+1: disable CDC from mysql to postgresql
10. T+2: start CDC sink from postgresql to mysql
11. T+5+: monitor saturation, errors, slow queries

Rollback criteria (must be explicit)

- Functional: failed business reconciliation
- Performance: sustained p99 regression above threshold
- Stability: elevated deadlocks / lock timeout spikes
- Data quality: checksum drift on critical entities
- Decide rollback window ahead of cutover day

PostgreSQL hardening after migration

- Autovacuum per-table scale factors for hot tables
- Aggressive `ANALYZE` cadence for volatile relations
- WAL/checkpoint tuning to reduce write stalls
- Backup validation: restore test + PITR verification
- Capacity model for 3-6 months growth

Observability dashboard minimum set

- `pg_stat_statements` : top total time, mean, stddev
- Wait events and lock trees
- Checkpoint timing and WAL generation rate
- Replication lag and slot health
- Bloat indicators and vacuum progress

Common post-cutover incidents

- Sequence behind max(id) after manual loads
- Missing or mismatched collations in search features
- ORM-generated SQL relying on MySQL implicit casts
- Long transactions blocking vacuum
- Over-indexing causing high write amplification

Final checklist

- Architecture selected by RTO/RPO, not preference
- Compatibility risks validated with real workload
- Reconciliation automated and rehearsed
- Cutover + rollback runbooks tested in staging
- Post-migration SLO ownership clearly assigned

Join Our DBA Team

Senior PostgreSQL DBA	Mid-level PostgreSQL DBA
	
https://www.linkedin.com/jobs/view/4412739500/	https://www.linkedin.com/jobs/view/4412746199/

Q&A

Thank you!

Stay Connected

LinkedIn	Presentation
www.linkedin.com/in/mehmet-emin-karakas	github.com/emin100/postgres_ankara_2026/releases/tag/latest