Application migration from SQL Server to PostgreSQL

Vladimir Kulikov

GitHub: YohDeadfall

Twitter: YohDeadfall

Agenda

- What is PostgreSQL?
- How is it working?
- How to migrate code?
- How to improve performance?

PostgreSQL

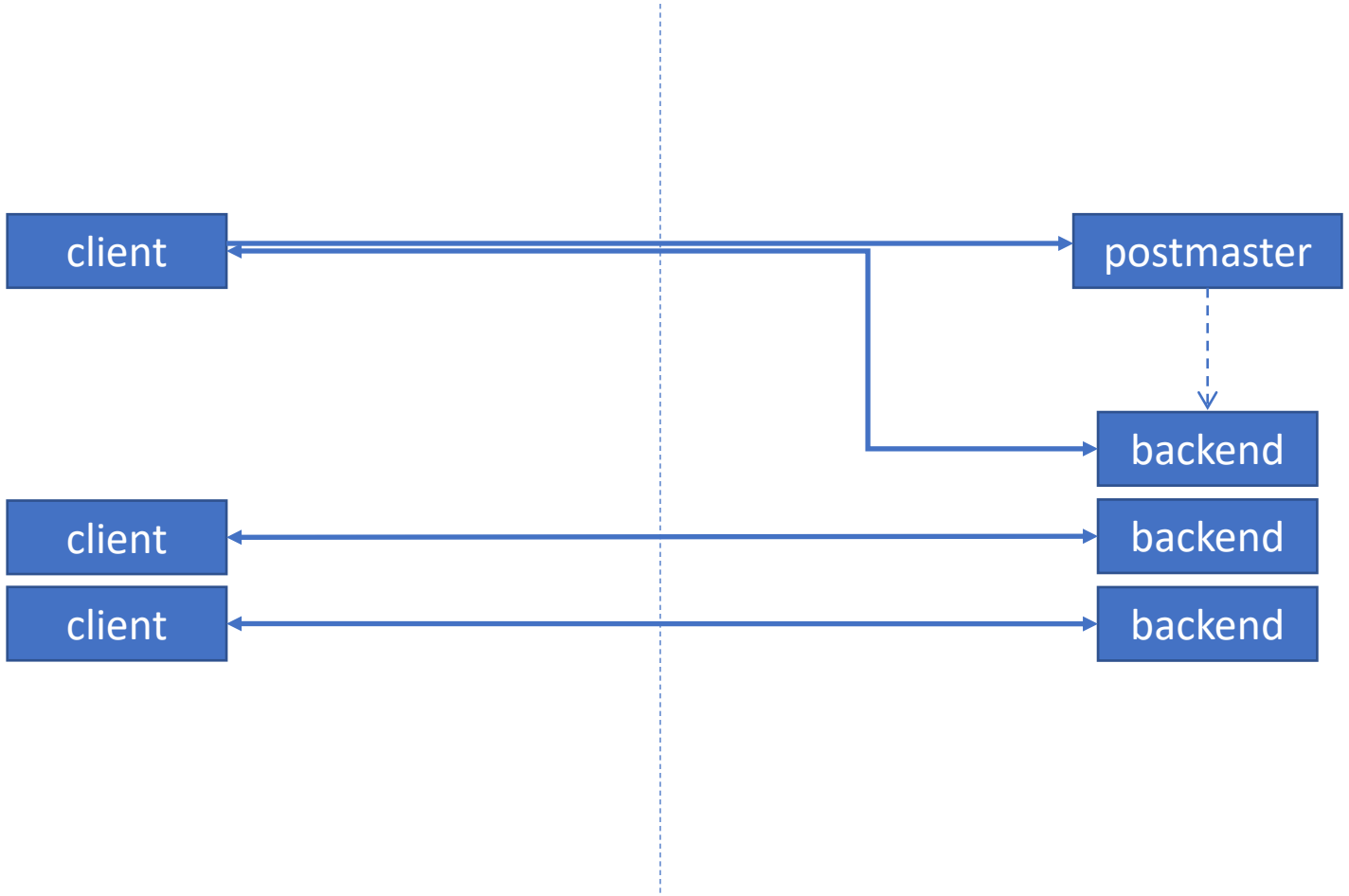
What is PostgreSQL

- Object relational database management system
- Over 20 years of history
- Open source
- Large community

Very simple process model

Client side

Server side



Code migration

What is Npgsql

- ADO.NET driver for PostgreSQL
- Open source
- 6th in TechEmpower Fortune (physical hardware)
- 3th in TechEmpower Fortune (cloud hardware)

Changes in code

Before:

```
using (var connection = new SqlConnection(connectionString))
using (var command = new SqlCommand("DROP TABLE students", connection))
{
   // ...
}
```

After:

```
using (var connection = new NpgsqlConnection(connectionString))
using (var command = new NpgsqlCommand("DROP TABLE students", connection))
{
   // ...
}
```

Is it all?
Likelihood no...

Limited connection count on the server side

The reasons of connection exhausting

- Application holds connection for a too long period
  - Bad application design
  - Business logic inside queries and functions
- PostgreSQL process model and low connection limit

Use connections only when necessary

Incorrect:

```
using (var connection = new NpgsqlConnection(connectionString))
using (var command = new NpgsqlCommand("DROP TABLE students", connection))
{
   await connection.OpenAsync(cancellationToken);
   // Command initialization goes here
   await command.ExecuteNonQueryAsync(cancellationToken);
}
```

Correct:

```
using (var connection = new NpgsqlConnection(connectionString))
using (var command = new NpgsqlCommand("DROP TABLE students", connection))
{
   // Command initialization should be here
   await connection.OpenAsync(cancellationToken);
   await command.ExecuteNonQueryAsync(cancellationToken);
}
```

Use pooling middleware

- pgBouncer

  Lightweight connection pooler that provides connection pooling.

- pgPool II

  Provides robust query routing and connection pooling for Postgres-based solutions.

- Odyssey

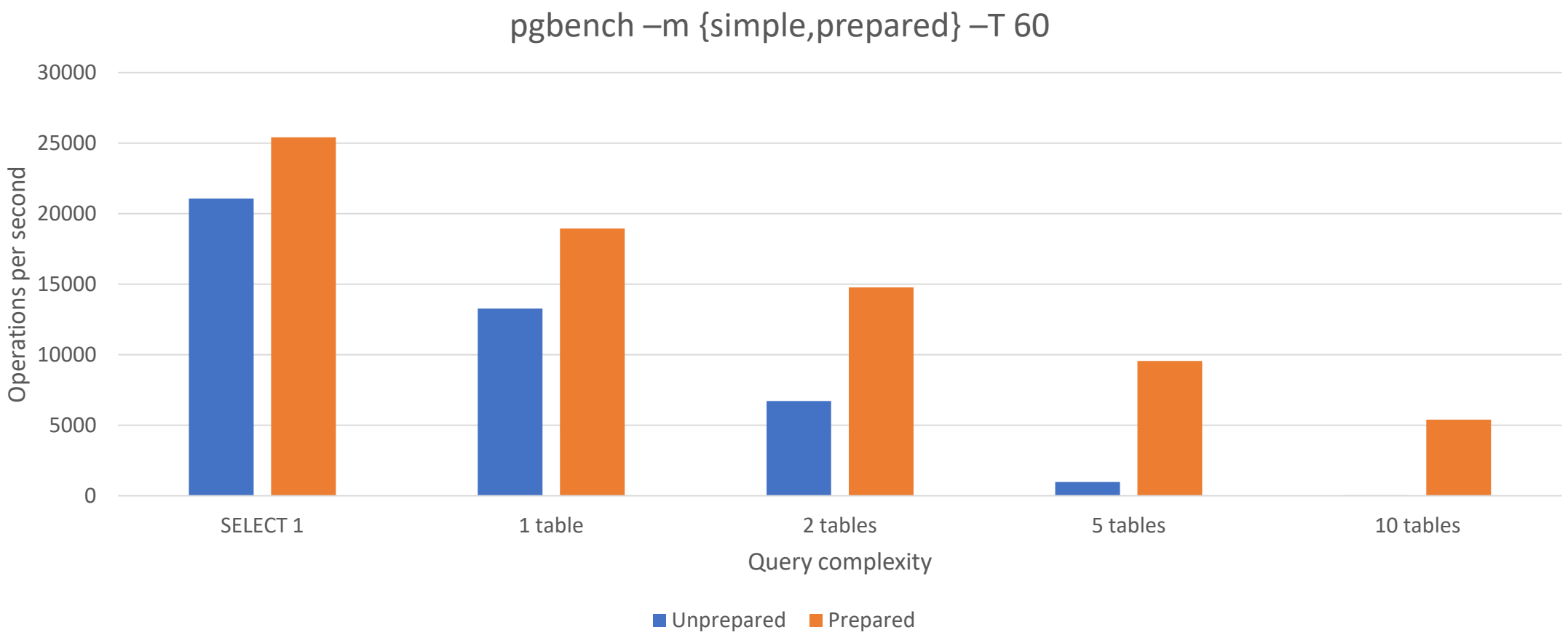
  Advanced multi-threaded PostgreSQL connection pooler and request router.

Limited query plan caching

The reasons of plan cache limitation

- No explicit statement preparation
- No plan caching between backend processes

Unprepared and prepared statements



Use prepared statements

- Manually via the Prepare method of NpgsqlCommand
- Automatically by setting the Max Auto Prepare parameter in the connection string

Use the rich type system for batch processing

Instead of multiple command execution:

```
var points = new [] { new Point(10, 20), new Point (30, 40) };
using (var command = new NpgsqlCommand("INSERT INTO points VALUES (@x, @y)", connection))
{
   var parameterX = command.Parameters.Add(new NpgsqlParameter(“x"));
   var parameterY = command.Parameters.Add(new NpgsqlParameter(“y"));
   foreach (var point in points)
   {
      parameterX.Value = point.X;
      parameterY.Value = point.Y;
      command.ExecuteNonQuery();
   }
}
```

Use the rich type system for batch processing

Execute it once with array of composites:

```
var points = new [] { new Point(10, 20), new Point (30, 40) };
using (var command = new NpgsqlCommand("INSERT INTO points SELECT * FROM @p", connection))
{
   command.Parameters.AddWithValue("p", points);
   command.ExecuteNonQuery();
}
```

How to use composites?

- Create the required type via CREATE TYPE
- Register type mapping in the application

SQL

```
CREATE TYPE point AS (x integer, y integer);
```

C# (global registration)

```
NpgsqlConnection.TypeMapper.MapComposite<Point>("point");
```

C# (per connection registration)

```
connection.TypeMapper.MapComposite<Point>("point");
```

Other performance improvements

Prefer generic methods and parameters

- Faster routing to the write method
- Low pressure on the GC

```
using (var command = new NpgsqlCommand(“SELECT @p", connection))
{
   command.Parameters.Add(new NpgsqlParameter<int>("p", 42));
   command.ExecuteNonQuery();
}
```

Batching/Pipelining

- Performs single roundtrip to the database
- Impact depends on the network latency

```
using (var cmd = new NpgsqlCommand("SELECT ...; SELECT ..."))
using (var reader = cmd.ExecuteReader())
{
   while (reader.Read()) { /* Read first resultset */ }
   reader.NextResult();
   while (reader.Read()) { /* Read second resultset */ }
}
```

Future of Npgsql

Future of Npgsql

- Performance improvements
  - Devirtualization of type handlers
  - Handling composites using dynamic methods
  - Pipelines/multiplexing
- Type handling improvements
  - ROW to ValueTuple mapping
  - Non-parameterless constructor support
- Monitoring and tracing via DiagnosticsSource

Thank you!