



DEVELOPER GUIDE

ElephantSqlDb.Data

ADO.NET Provider and OrmLite
Dialect for ElephantSqlDB®

Raw ADO.NET · OrmLite · Dapper
Databases, schema, paging and file records
NuGet: ElephantSqlDb.Data · .NET 8

ElephantSqlDB, Inc. · dba Dataark Systems, LLC · Austin, Texas
U.S. Patent Application No. 63/974,005 · NVIDIA Inception Program Member · AWS Marketplace



ElephantSqlDb.Data — Developer Guide

ADO.NET provider and ServiceStack.OrmLite dialect for ElephantSqlDB.

Version 0.1.0-alpha. Written against ElephantSqlDbClient 1.0.188 and the product documentation. Not yet exercised against a live table — work through [Verification](#) before depending on it.

Contents

1. [What this is](#)
 2. [Architecture](#)
 3. [Installation](#)
 4. [Connection strings](#)
 5. [Dependency injection](#)
 6. [Raw ADO.NET](#)
 7. [Pagination and result metadata](#)
 8. [OrmLite](#)
 9. [Dapper](#)
 10. [Type mapping](#)
 11. [No transactions](#)
 12. [No DDL, no constraints](#)
 13. [Parameters and injection](#)
 14. [Performance](#)
 15. [Error handling](#)
 16. [Testing](#)
 17. [Verification](#)
 18. [Migrating from PostgreSQL](#)
 19. [Troubleshooting](#)
-

1. What this is

ElephantSqlDbClient.Core is an HTTP/WebSocket API client with its own shape: operations are extension methods split by statement type, results come back generic over a CLR type, and the table is an argument on every call.

This package wraps that client in the ADO.NET abstractions — `DbConnection`, `DbCommand`, `DbDataReader`, `DbParameter` — so anything that speaks ADO.NET can reach an ElephantSqlDB table without knowing the client exists. That means OrmLite, Dapper, EF Core, reporting tools, and any code already written against `IDbConnection`.

It also ships an OrmLite dialect, so `IDbConnectionFactory` registration is a one-line change from a PostgreSQL-backed application.

Where the client returns more than rows — the `PaginationDTO` and `AuthStatus` that come back from every `SELECT` — the provider keeps them reachable rather than discarding them at the ADO.NET boundary. See [section 7](#).

What it does not do is paper over gaps in the underlying engine. There are no transactions, no DDL, and no enforced constraints. The provider surfaces those as exceptions rather than silently pretending — see sections 11 and 12, which are the two you should read before writing any code.

2. Architecture



Files

File	Role
ElephantConnection.cs	DbConnection, DbTransaction, DbProviderFactory
ElephantCommand.cs	DbCommand
ElephantDataReader.cs	DbDataReader over paged, untyped rows
ElephantParameter.cs	DbParameter, collection, and the literal binder
ElephantConnectionStringBuilder.cs	Connection string keywords
IElephantExecutor.cs	Seam between ADO.NET and the client
ElephantSqlDbExecutor.cs	Adapter over ElephantSqlDbClient.Core
ElephantConnectionExtensions.cs	Paged query helpers
ElephantOrmLiteDialectProvider.cs	OrmLite dialect and type converters

How a query flows

1. ElephantCommand.ExecuteReaderAsync validates state.
2. ElephantParameterBinder renders parameter values into the SQL text — the backend binds nothing, so substitution is the only route.
3. ElephantSqlDbExecutor.QueryAsync calls SelectSQLRecordsAsync<string>. T is string deliberately: the client special-cases it and returns raw JSON rather than mapping onto a CLR type the provider does not have.
4. JSON rows are projected into IDictionary<string, object?>.
5. The client's PaginationDTO and AuthStatus are captured as an ElephantPagination, so the caller can reach TotalItems and the outcome status rather than only the rows.
6. ElephantDataReader fixes column names and ordinals from the first page and converts values on demand.

How a write flows

ElephantSqlDbExecutor.ExecuteAsync classifies the leading keyword and routes:

Statement	Route
INSERT	Parsed, then SaveToDatabaseWithNoResponse once per VALUES row; the target database travels in the record envelope's DatabaseName
UPDATE	UpdateSQLRecordsAsync with DatabaseName
DELETE	DeleteSQLRecordsAsync with DatabaseName
SAVEFILEASRECORD	SaveSQLRecordsFromFileAsync with DatabaseName

Statement	Route
CREATE TABLE	Accepted and dropped — tables appear on first write
CREATE DATABASE [IF NOT EXISTS] name	CreateDatabaseAsync — see Creating a database
ALTER, DROP, TRUNCATE, CREATE INDEX	NotSupportedException
Anything else	ExecuteRootKeyCommandAsync

Inserts bypass the client's InsertSQLRecordsAsync on purpose. See [section 19](#) for why.

3. Installation

```
dotnet add package ElephantSqlDb.Data
```

Or reference the project directly:

```
<ProjectReference Include="..\ElephantSqlDb.Data\ElephantSqlDb.Data.csproj" />
```

Dependencies: ElephantSqlDbClient 1.0.188+, ServiceStack.OrmLite 8.x, Newtonsoft.Json 13.x. Targets net8.0.

Pin the OrmLite version rather than floating it. The dialect base class changed substantially in v8 — converters replaced the *ColumnDefinition fields, and TableRef replaced the (tableName, schema) pairs — so a floating major would break the overrides without warning.

4. Connection strings

```
BaseUri=https://api.elephantsql.db.example/;Token=...;Roles=Admin;UserEmail=svc@example.com;TenantId=0;Database=portal_table
```

Keyword	Required	Default	Notes
BaseUri	yes	—	Root of the HTTP API
Token	yes	—	Bearer token
UserEmail	yes	—	
Database	yes	—	The table the connection authenticates against, and the default database for every statement. Override per connection or per command — see Choosing the database
Roles	no	—	
TenantId	no	0	
TablePreface	no	Crypto-	
MaxRecords	no	100	Server-side page size
UseWebSockets	no	false	
MaxConnections	no	10	
AsyncScale	no	1	
CommandTimeout	no	30	Seconds
AllowNonTransactional	no	false	See section 11
AutoRegisterUser	no	false	Create the account on open — see below
AutoCreateTable	no	false	Create the table on open — see below

Keyword	Required	Default	Notes
Password	no	—	Only used by AutoRegisterUser
FirstName	no	Service	Registration detail
MiddleName	no	—	Registration detail
LastName	no	Account	Registration detail
Phone	no	—	Registration detail
EncryptionAlgorithm	no	SHA256	Table created by AutoCreateTable, and by CreateDatabaseAsync when it is not given one

Missing required keywords throw `InvalidOperationException` on `Open`, not on construction — a builder can be assembled incrementally.

Keeping the token out of source

Leave it blank in `appsettings.json` and supply it from the environment. Anything that reads `IConfiguration` picks it up through the normal precedence chain.

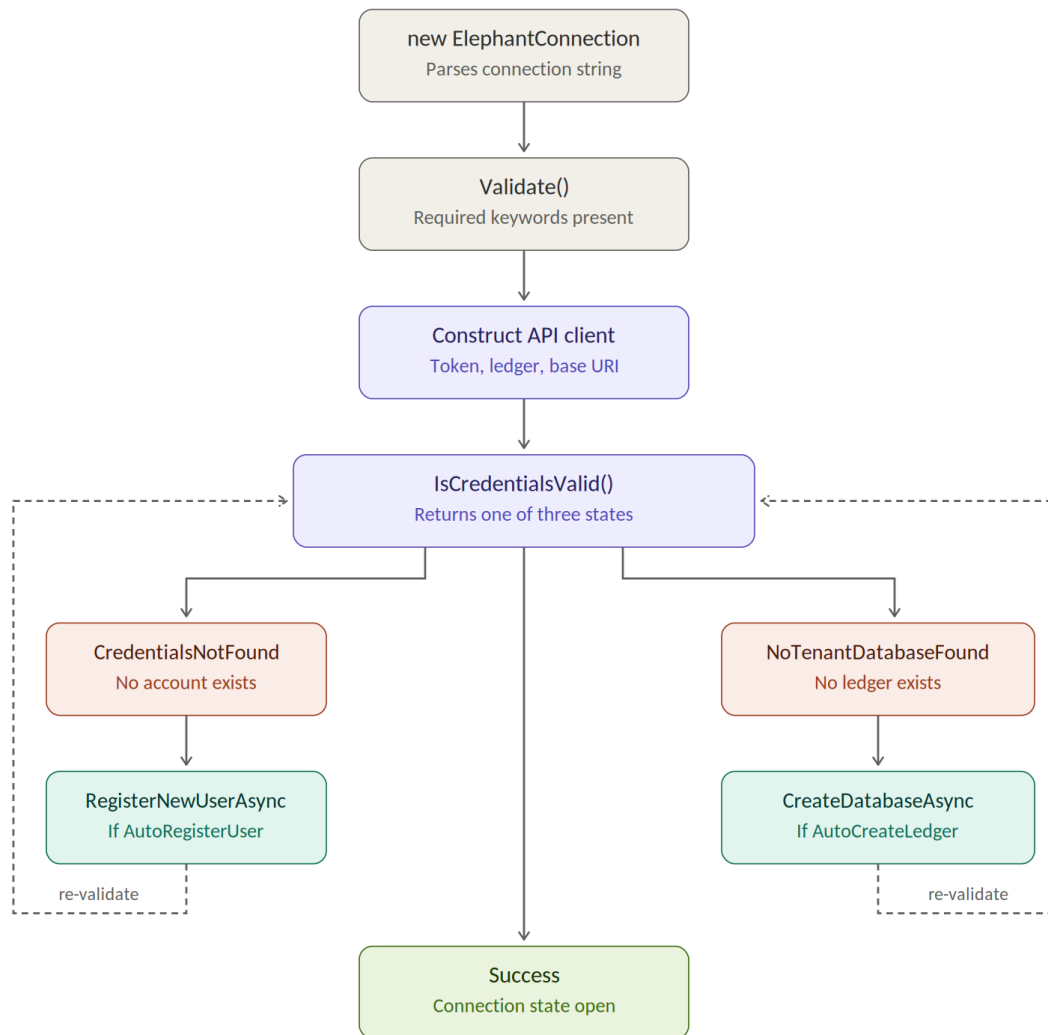
```
{
  "ConnectionStrings": {
    "Portal":
"BaseUri=https://api.elephantsql.com/example/;Token=;Roles=;UserEmail=;Database=portal_table;MaxRecords=250"
  }
}
```

```
ConnectionStrings__Portal=BaseUri=...;Token=real-token;...
```

In Kubernetes, source that variable from a secret rather than baking it into the image.

What happens on Open

`OpenAsync` does more than dial the API. It parses and validates the connection string, constructs the client, then calls `IsCredentialsValid` — and two of the three states that call can return are recoverable.



The two side branches are each gated by a connection string flag. With the flag off the branch throws instead of remediating, and the message names the flag to set. Both loops return to `IsCredentialsValid` rather than proceeding directly, and the whole sequence is bounded to two passes — at most one registration and one table creation, so a server that keeps reporting the same recoverable state fails with that state rather than spinning.

The three states:

Status	Meaning	Provider behaviour
Success	Account and table both exist	Connection opens
CredentialsNotFound	No account for this email	Registers it if <code>AutoRegisterUser=true</code> , otherwise throws
NoTenantDatabaseFound	Account exists, table does not	Creates it if <code>AutoCreateTable=true</code> , otherwise throws

Concurrency is not a concern. The engine is actor-based and message driven, so simultaneous requests for the same table resolve to the same entity— the first message creates it, later ones address the one that exists. There is no duplicate-create race, which is why a reported error is treated as a real failure rather than something to re-validate past: a message from `CreateDatabaseAsync` means a rejected name, a quota, or a permission, and the provider throws with it.

Each remediation is still followed by a fresh `IsCredentialsValid`, confirming the state it was meant to reach was actually reached.

Both flags are **off by default**, and the exception when they are off tells you which one to set:

```
ElephantException: Table 'portal_table' does not exist for 'svc@example.com'.
Create it in the management portal, or set AutoCreateTable=true to have the
provider create it on open.
```

When to turn them on

`AutoCreateTable` is reasonable for local development, integration tests and throwaway environments. Repeated creation is harmless — the entitymodel makes it idempotent — but a typo in `Database` produces an empty table instead of an error, and the application then runs against nothing.

`AutoRegisterUser` should stay off anywhere that is not a developer machine. Opening a connection is not the place to create an identity: it needs the password in the connection string, it races across replicas, and a stale `UserEmail` variable becomes a new account rather than a failure. Provision service accounts deliberately.

```
// Local development - stand everything up on first run.
var dev = new ElephantConnectionStringBuilder
{
    BaseUri = "https://api.elephantsql.db.example/",
    Token = token,
    UserEmail = "dev@example.com",
    Database = "dev_table",
    AutoRegisterUser = true,
    Password = "...",
    AutoCreateTable = true
};

// Production - fail loudly if either is missing.
var prod = new ElephantConnectionStringBuilder
{
    BaseUri = "https://api.elephantsql.db.example/",
    Token = token,
    UserEmail = "svc@example.com",
    Database = "portal_table"
};
```

Doing it yourself

Where the bootstrap needs to be explicit — a provisioning job rather than a connection open — drive the client directly and leave both flags off:

```
var client = factory.GetApiClient();
var status = await client.IsCredentialsValid();

if (status == AuthStatus.CredentialsNotFound)
{
    await client.RegisterNewUserAsync(
        email, password, firstName, middleName, lastName, phone, false);

    status = await client.IsCredentialsValid();
}

if (status == AuthStatus.NoTenantDatabaseFound)
{
    var (errorMessage, _) = await client.CreateDatabaseAsync(client.GetTableName());

    if (!string.IsNullOrEmpty(errorMessage))
        logger.LogWarning("Table creation reported: {Message}", errorMessage);

    status = await client.IsCredentialsValid();
}
```

```
if (status != AuthStatus.Success)
    throw new InvalidOperationException($"Provisioning did not complete: {status}.");
```

Building one in code

```
var builder = new ElephantConnectionStringBuilder
{
    BaseUri = "https://api.elephantsql.com/",
    Token = token,
    UserEmail = "svc@example.com",
    Database = "portal_table",
    MaxRecords = 250,
    CommandTimeout = 60
};

builder.Validate();

var connectionString = builder.ConnectionString;
```

5. Dependency injection

Program.cs — minimal hosting

```
using ElephantSqlDb.Data;
using ServiceStack.Data;
using ServiceStack.OrmLite;

var builder = WebApplication.CreateBuilder(args);

var connectionString = builder.Configuration.GetConnectionString("Portal")
?? throw new InvalidOperationException(
    "ConnectionStrings:Portal is not set. In the cluster this comes from the " +
    "ConnectionStrings__Portal environment variable.");

// Singleton. The factory holds no connection - it hands one out per
// OpenDbConnectionAsync call.
builder.Services.AddSingleton<IDbConnectionFactory>(_ => new OrmLiteConnectionFactory(
    connectionString,
    ElephantOrmLiteDialectProvider.Instance));

var app = builder.Build();
app.Run();
```

Startup.cs — classic hosting

```
public class Startup(IConfiguration configuration)
{
    public void ConfigureServices(IServiceCollection services)
    {
        var connectionString = configuration.GetConnectionString("Portal")
        ?? throw new InvalidOperationException("ConnectionStrings:Portal is not set.");

        services.AddSingleton<IDbConnectionFactory>(_ => new OrmLiteConnectionFactory(
            connectionString,
            ElephantOrmLiteDialectProvider.Instance));
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env) { }
}
```

Raw ADO.NET without OrmLite

```
builder.Services.AddSingleton(new ElephantConnectionStringBuilder
{
```

```

        connectionString = connectionString
    });

builder.Services.AddScoped<DbConnection>(sp =>
    new ElephantConnection(
        sp.GetRequiredService<ElephantConnectionStringBuilder>().ConnectionString));

```

Scoped, not singleton — DbConnection is not thread-safe, and a singleton would be shared across concurrent requests.

Registering by invariant name

```

ElephantProviderFactory.Register();

var factory = DbProviderFactories.GetFactory("ElephantSqlDb.Data");
using var connection = factory.CreateConnection();

```

Useful for tools that resolve providers by name.

Do you need a repository per table?

No. IDbConnectionFactory is the only registration the provider requires. Everything above it is your choice of structure. Three reasonable shapes:

Inject the factory directly. OrmLite is already a data-access abstraction; wrapping it per table mostly adds indirection.

```

public sealed class OfferingService(IDbConnectionFactory dbFactory)
{
    public async Task<List<Offering>> GetOpenAsync(CancellationToken ct = default)
    {
        using var db = await dbFactory.OpenDbConnectionAsync(token: ct);
        return await db.SelectAsync<Offering>(o => o.Status == OfferingStatus.Open, token: ct);
    }
}

```

One open generic, registered once.

```

services.AddScoped(typeof(IRepository<>), typeof(Repository<>));

```

Repositories only where they earn it. Under this provider there is a specific argument for this: the engine enforces no unique constraints, so checks like "is this slug taken" have to live somewhere consistent. A repository is a reasonable home. That is a handful of aggregates, not one per table.

6. Raw ADO.NET

Query

```

await using var connection = new ElephantConnection(connectionString);
await connection.OpenAsync();

await using var command = connection.CreateCommand();
command.CommandText = "SELECT id, name, close_at FROM offerings WHERE status = @status";
command.Parameters.Add(new ElephantParameter("status", "Open"));

await using var reader = await command.ExecuteReaderAsync();

while (await reader.ReadAsync())
{
    var id = reader.GetGuid(0);
    var name = reader.GetString(1);
    var closeAt = reader.IsDBNull(2) ? (DateTime?)null : reader.GetDateTime(2);
}

```

Scalar

```
await using var command = connection.CreateCommand();
command.CommandText = "SELECT COUNT(*) AS total FROM subscriptions WHERE offering_id = @id";
command.Parameters.Add(new ElephantParameter("id", offeringId));

var total = Convert.ToInt64(await command.ExecuteScalarAsync());
```

Returns the first column of the first row, or null on an empty result.

Write

```
await using var command = connection.CreateCommand();
command.CommandText =
    "INSERT INTO offerings (id, slug, name, status) VALUES (@id, @slug, @name, @status)";

command.Parameters.Add(new ElephantParameter("id", Guid.NewGuid()));
command.Parameters.Add(new ElephantParameter("slug", "seed-round"));
command.Parameters.Add(new ElephantParameter("name", "Seed Round"));
command.Parameters.Add(new ElephantParameter("status", "Open"));

var affected = await command.ExecuteNonQueryAsync();
```

ExecuteNonQuery returns a real row count for INSERT, because the provider loops the rows itself. For UPDATE and DELETE it returns -1 — ADO.NET's "not known" — because the API reports no count and a fabricated 1 would be worse than none.

Saving a file as a record

SAVEFILEASRECORD stores a file — an ONNX model, a document, an image — as a record. Pass it as the command text and run it with ExecuteNonQuery:

```
await using var connection = new ElephantConnection(connectionString);
await connection.OpenAsync();
connection.ChangeDatabase("postgresql_postgres_public");

await using var command = connection.CreateCommand();
command.CommandText =
    @"SAVEFILEASRECORD ""C:\Dataark Systems LLC\Sourcecode\ESP-
FailuresModel\onnx_models\esp_pump\esp_failure_model.onnx"", ""esp_failures_v001""";

await command.ExecuteNonQueryAsync(); // -1 on success; throws ElephantException otherwise
```

It is the client-side equivalent of:

```
AuthStatus status = await client.SaveSQLRecordsFromFileAsync(sqlFile, DatabaseName:
"postgresql_postgres_public");
```

- The first argument is the file path, the second the record name; both in double quotes. A per-command DatabaseName works here as it does for any statement.
- The path is read by the client on the machine running your code, not by the server.
- Write the path and name into the command text. Parameters render as single-quoted SQL literals, which is not the form this command takes; @ inside a double-quoted path is left alone.
- ExecuteNonQuery returns -1: the API reports a status, not a count.

Reading by column name

```
while (await reader.ReadAsync())
{
    var id = reader.GetGuid(reader.GetOrdinal("id"));
    var name = reader["name"] as string;
}
```

Column lookup is case-insensitive. GetOrdinal throws IndexOutOfRangeException for an unknown name.

Choosing the database

Every statement carries a database name, the same `DatabaseName` argument the client takes on `SelectSQLRecordsAsync`, `InsertSQLRecordsAsync` and the rest. By default it is the `Database` keyword. It can be overridden at two levels:

Level	How	Scope
Connection	<code>connection.ChangeDatabase("name")</code>	Every later command on that connection
Command	<code>command.DatabaseName = "name"</code>	That command only

The most specific setting wins: `command.DatabaseName`, then `ChangeDatabase`, then the `Database` keyword. Authentication is unaffected — Open always validates against the `Database` keyword, and an override only changes where statements run.

Per connection — the equivalent of passing the same `DatabaseName` to every client call:

```
await using var connection = new ElephantConnection(connectionString); // Database=dataaDatabase
await connection.OpenAsync();

connection.ChangeDatabase("postgresql_postgres_public");

await using var insert = connection.CreateCommand();
insert.CommandText = "INSERT INTO api_key (id, user_auth_id, created_date) VALUES (@id, @user, @created)";
insert.Parameters.Add(new ElephantParameter("id", ApiKey.NewId()));
insert.Parameters.Add(new ElephantParameter("user", userAuthId));
insert.Parameters.Add(new ElephantParameter("created", DateTime.UtcNow));
await insert.ExecuteNonQueryAsync(); // written to postgresql_postgres_public

await using var select = connection.CreateCommand();
select.CommandText = "SELECT id, created_date FROM api_key WHERE user_auth_id = @user";
select.Parameters.Add(new ElephantParameter("user", userAuthId));
await using var reader = await select.ExecuteReaderAsync(); // read from postgresql_postgres_public
```

It is the client-side equivalent of:

```
await client.InsertSQLRecordsAsync<string>(sql, DatabaseName: "postgresql_postgres_public");
await client.SelectSQLRecordsAsync<string>(sql, DatabaseName: "postgresql_postgres_public");
```

Per command — one statement against a different database, the rest of the connection unchanged:

```
await using var command = connection.CreateCommand();
command.CommandText = "SELECT id, name FROM offerings WHERE status = @status";
command.DatabaseName = "postgresql_postgres_public_Wg6Jv1Nf18";
command.Parameters.Add(new ElephantParameter("status", "Open"));

await using var reader = await command.ExecuteReaderAsync();
```

Leaving `DatabaseName` null (or blank) falls back to the connection.

Reading the current value — `connection.Database` returns the override if one is set, otherwise the `Database` keyword:

```
Assert.Equal("dataaDatabase", connection.Database);
connection.ChangeDatabase("postgresql_postgres_public");
Assert.Equal("postgresql_postgres_public", connection.Database);
```

Through Dapper and OrmLite — both sit on the same connection, so set it there before running anything:

```
// Dapper: IDbConnection.ChangeDatabase
await using var connection = new ElephantConnection(connectionString);
await connection.OpenAsync();
connection.ChangeDatabase("postgresql_postgres_public");
var keys = await connection.QueryAsync<ApiKey>("SELECT * FROM api_key WHERE user_auth_id = @u", new { u =
userAuthId });
```

```
// Ormlite: unwrap to the provider's connection first
using var db = await dbFactory.OpenDbConnectionAsync();
db.ToDbConnection().ChangeDatabase("postgresql_postgres_public");
var rows = await db.SelectAsync<ApiKey>(k => k.UserAuthId == userAuthId);
```

Things to know:

- ChangeDatabase rejects a null or blank name with ArgumentException.
- The override lasts until it is changed again or the connection string is replaced; assigning ConnectionString clears it. It is not part of the connection string, so a pooled or factory-created connection starts on the Database keyword.
- Nothing is validated on the server when the name is set. A misspelt name surfaces on the first statement — as an empty result for a SELECT, or as rows written where you did not expect them for an INSERT.
- INSERT still goes through SaveToDatabaseWithNoResponse with decoded values; the target database is carried in the record envelope's DatabaseName, which the server uses to route the save. UPDATE, DELETE and other statements pass it as the DatabaseName argument.

Listing databases and their schema

GetDatabasesAsync returns the databases the connection's account can see, each with its full schema, read from the client's DatabaseModel records and sorted by display name.

Name is the database's identity — the value ChangeDatabase, command.DatabaseName and the table use — and it never changes. DisplayName is for display only; see [Renaming a database](#).

Provider record	From	Fields
ElephantDatabase	DatabaseModel	Name, DisplayName, Tables
ElephantTable	TableModel	Name, Columns
ElephantColumn	ColumnModel	Name, DataType, IsPrimaryKey, IsForeignKey, ForeignKey
ElephantForeignKey	ForeignKeyModel	Name, ReferencedTable, ReferencedColumn

```
await using var connection = new ElephantConnection(connectionString);
await connection.OpenAsync();

foreach (var db in await connection.GetDatabasesAsync())
{
    Console.WriteLine($"{db.DisplayName} ({db.Name})");

    foreach (var table in db.Tables)
        Console.WriteLine($"  {table.Name}: {string.Join(", ", table.Columns.Select(c => $"{c.Name} {c.DataType}"))}");
}

// One database, by Name (defaults to connection.Database)
var schema = await connection.GetDatabaseAsync("postgresql_postgres_public");
connection.ChangeDatabase(schema!.Name);
```

- DisplayName is DatabaseDisplayName, or Name when that is blank. Entries are keyed by Name; if two records share a Name, the first wins.
- With no DatabaseModel records, GetDatabasesAsync returns a single entry, with no tables, for connection.Database. Pass includeCurrentWhenEmpty: false for an empty list. GetDatabaseAsync matches on Name only and returns null for an unknown name.
- It runs select * from DatabaseModel where name != "" typed as DatabaseModel, with no DatabaseName and StringNameOfType: "DatabaseModel", reading every page. It is not affected by ChangeDatabase, and each call is one fresh listing — cache the result if you call it often.

The same data is available through the standard ADO.NET schema API, for tools that discover schema that way. Restrictions are positional; null or empty means any. The database restriction matches Name.

Collection	Columns	Restrictions
Databases	database_name, display_name	—
Tables	database_name, table_name	database, table
Columns	database_name, table_name, column_name, ordinal_position, data_type, is_primary_key, is_foreign_key	database, table, column
ForeignKeys	database_name, table_name, column_name, constraint_name, referenced_table, referenced_column	database, table

```
DataTable databases = await connection.GetSchemaAsync("Databases");
DataTable tables   = await connection.GetSchemaAsync("Tables", ["postgresql_postgres_public"]);
DataTable columns  = await connection.GetSchemaAsync("Columns", ["postgresql_postgres_public",
"api_key"]);
```

GetSchema() with no arguments returns MetaDataCollections, listing these four.

Creating a database

CreateDatabaseAsync creates a database (table) for the connection's account, on an open connection, independently of AutoCreateTable:

```
await using var connection = new ElephantConnection(connectionString);
await connection.OpenAsync();

await connection.CreateDatabaseAsync("sales_2026");           // EncryptionAlgorithm from the
connection string
await connection.CreateDatabaseAsync("audit_vault", "SHA512"); // explicit algorithm

connection.ChangeDatabase("sales_2026");                      // start using it
```

The SQL form runs through any command, which is how tools that only speak SQL can do it:

```
await using var command = connection.CreateCommand();
command.CommandText = "CREATE DATABASE IF NOT EXISTS sales_2026";
await command.ExecuteNonQuery(); // -1: status, not a count
```

- The name may contain letters, digits, _ and - (1–128 characters); anything else throws ArgumentException. In SQL it may be bare or wrapped in "...", [...] or `...`.
- It creates the table a connection with Database=<name> would open: the name is composed by the client's GetTableName(), exactly as for AutoCreateTable.
- The connection keeps its current database. Call ChangeDatabase, or set command.DatabaseName, to use the new one.
- A refusal from the server (a rejected name, a quota, a permission) throws ElephantException.
- IF NOT EXISTS runs one lookup — select * from DatabaseModel where name = "<name>" — and does nothing when a database with that Name exists. It does not list every database, and a display name never counts as a match. Without IF NOT EXISTS, the request goes to the server, which decides what an existing name means. The check and the create are two calls, not one atomic step.
- CreateDatabase(name) is the synchronous form.

Renaming a database

RenameDatabaseAsync changes a database's **display name only**. Name — the identity used by ChangeDatabase, command.DatabaseName and the table — is never changed, so existing connection strings, commands and stored rows keep working.

```
await connection.RenameDatabaseAsync("sales_2026", "Sales FY2026");
```

```
var db = await connection.GetDatabaseAsync("sales_2026"); // still found by Name
Console.WriteLine(db!.DisplayName); // "Sales FY2026"
connection.ChangeDatabase("sales_2026"); // Name, not the display name
```

- It reads the DatabaseModel whose Name matches (one lookup), sets DatabaseDisplayName, clears each table's SelectedNodeTable and Link so the model serializes, and saves it over its own record with SaveToDatabaseWithNoResponse (BlockName as the record id, its current BlockHashCode as the existing hash) — the same steps as the management tooling's SaveRenamedDatabaseContext.
- It returns the number of DatabaseModel records updated, normally 1. Name is not enforced unique, so every record carrying it is updated.
- An unknown Name throws ElephantException (NoDataFound). A display name must be 1–200 characters with no control characters.
- There is no SQL form. RenameDatabase(name, displayName) is the synchronous form.

7. Pagination and result metadata

SelectSQLRecordsAsync returns three things — status, rows, and a PaginationDTO. Most ADO.NET consumers map the rows onto objects and discard everything else, which loses TotalItems: the one value a paged UI needs and cannot compute for itself.

The provider keeps all three. ElephantPagination carries the DTO's fields plus the outcome status:

Member	Notes
PageNumber	1-based page this result came from
PageSize	Rows per page, as reported by the server
TotalItems	Rows matching the query across every page
TotalPages	Computed from TotalItems and PageSize
HasMore	A further page exists
IsEmpty	The query matched nothing
Status	AuthStatus name for the call — Success or NoDataFound

Status is a string rather than the client's enum so that IElephantExecutor stays free of the client dependency and the tests can run without it. The typed AuthStatus is on ElephantException for the failure path.

Reading it from the reader

Available before any row is consumed, because TotalItems comes back with the first page:

```
await using var command = connection.CreateCommand();
command.CommandText = "SELECT id, name FROM offerings WHERE status = @status";
command.Parameters.Add(new ElephantParameter("status", "Open"));

await using var reader = (ElephantDataReader)await command.ExecuteReaderAsync();

var total = reader.TotalItems; // 412
var pages = reader.Pagination.TotalPages; // 17 at PageSize 25
var more = reader.HasMorePages; // true
var status = reader.Pagination.Status; // "Success"

while (await reader.ReadAsync())
{
    // ...
}
```

Reading it from the command

LastPagination is set by ExecuteReader, ExecuteScalar and ExecuteNonQuery, and survives the reader being disposed — useful when a total is needed after the rows have been consumed:

```
using (await command.ExecuteReaderAsync()) { /* ... */ }  
var total = command.LastPagination!.TotalItems;
```

Server-side paging

Setting PageNumber fetches exactly that page and stops the reader walking forward. Without it the reader pages ahead as the consumer reads, which quietly turns a paged query into a full scan:

```
command.PageNumber = 3; // exactly page 3, no look-ahead
```

QueryPageAsync wraps that into one call:

```
var page = await connection.QueryPageAsync(  
    "SELECT id, name FROM offerings WHERE status = @status",  
    r => new OfferingSummary(r.GetGuid(0), r.GetString(1)),  
    pageNumber: 2,  
    parameters: [new ElephantParameter("status", "Open")]);  
  
page.Rows; // IReadOnlyList<OfferingSummary>  
page.PageNumber; // 2  
page.TotalItems; // 412  
page.TotalPages; // 17  
page.HasMore; // true
```

Counting without materialising

TotalItems arrives with the first page, so a count is one round trip and no COUNT(*):

```
var total = await connection.CountAsync(  
    "SELECT id FROM subscriptions WHERE offering_id = @id",  
    [new ElephantParameter("id", offeringId)]);
```

Streaming every page

```
await foreach (var name in connection.QueryAllAsync(  
    "SELECT name FROM offerings", r => r.GetString(0), cancellationToken: ct))  
{  
    // ...  
}
```

Each page is a round trip. Bound the query or raise MaxRecords rather than streaming a whole table.

Through an interface-typed call

OrmLite and Dapper hand back IDbCommand and DbDataReader, so the concrete types are erased. Two extension methods recover the pagination:

```
ElephantPagination? fromCommand = someDbCommand.GetPagination();  
ElephantPagination? fromReader = someDbDataReader.GetPagination();
```

Both return null when the object did not come from this provider.

Empty results

An empty result reports NoDataFound, not Success. That distinction matters: it separates "nothing matched the query" from "this page happened to be short."

```
if (reader.Pagination.IsEmpty)
{
    // No rows anywhere, not just on this page.
}
```

8. OrmLite

Entities

```
[Alias("offerings")]
public class Offering
{
    [PrimaryKey, Alias("id")]
    public Guid Id { get; set; } = Guid.NewGuid();

    [Alias("slug"), StringLength(120)]
    public required string Slug { get; set; }

    [Alias("name"), StringLength(200)]
    public required string Name { get; set; }

    [Alias("status")]
    public OfferingStatus Status { get; set; }

    [Alias("target_amount"), DecimalLength(18, 2)]
    public decimal TargetAmount { get; set; }

    [Alias("close_at")]
    public DateTimeOffset? CloseAt { get; set; }

    [Alias("created_at")]
    public DateTimeOffset CreatedAt { get; set; } = DateTimeOffset.UtcNow;
}
```

Attributes that behave differently here:

- [Schema] is **ignored**. The table on the connection is the schema, so portal.offerings would name nothing.
- [Index] and [Index(Unique = true)] are **read and ignored**. No index DDL is emitted and no constraint is created.
- [AutoIncrement] **throws**. There is no sequence or identity — assign keys yourself.
- [ForeignKey] produces no constraint.

Reads

```
using var db = await dbFactory.OpenDbConnectionAsync(token: ct);

var open = await db.SelectAsync<Offering>(
    o => o.Status == OfferingStatus.Open, token: ct);

var one = await db.SingleAsync<Offering>(o => o.Slug == slug, token: ct);

var byId = await db.SingleByIdAsync<Offering>(id, token: ct);

var exists = await db.ExistsAsync<Offering>(o => o.Slug == slug, token: ct);
```

Typed expressions

```
var query = db.From<Offering>()
    .Where(o => o.Status == OfferingStatus.Open)
    .And(o => o.CloseAt > DateTimeOffset.UtcNow)
    .OrderByDescending(o => o.CreatedAt)
    .Limit(skip: 0, rows: 25);
```

```
var page = await db.SelectAsync(query, token: ct);
```

Limit becomes LIMIT ... OFFSET Note this sits *on top of* the client's own pagination: the API returns pages of MaxRecords regardless, and the reader walks them. A LIMIT bounds the logical result; it does not make the fetch one round trip.

Writes

```
offering.Id = Guid.NewGuid();
offering.CreatedAt = DateTimeOffset.UtcNow;

await db.InsertAsync(offering, token: ct);

offering.Name = "Seed Round (Extended)";
await db.UpdateAsync(offering, token: ct);

await db.DeleteByIdAsync<Offering>(offering.Id, token: ct);
```

Prefer explicit InsertAsync/UpdateAsync over SaveAsync. OrmLite's upsert paths may reach for a transaction, which throws unless AllowNonTransactional=true.

Raw SQL through OrmLite

```
var rows = await db.SelectAsync<Offering>(
    "SELECT * FROM offerings WHERE status = @status",
    new { status = "Open" }, token: ct);

var total = await db.ScalarAsync<long>(
    "SELECT COUNT(*) FROM subscriptions WHERE offering_id = @id",
    new { id = offeringId }, token: ct);
```

Schema calls

```
// Succeeds and does nothing. The table appears on the first insert.
await db.CreateTableIfNotExistsAsync<Offering>();

// Real: uses information_schema, which the engine supports.
var exists = db.TableExists<Offering>();
```

Leaving CreateTableIfNotExists in your startup is harmless, but understand that it is not provisioning anything.

9. Dapper

```
await using var connection = new ElephantConnection(connectionString);
await connection.OpenAsync();

var open = await connection.QueryAsync<Offering>(
    "SELECT id, slug, name FROM offerings WHERE status = @status",
    new { status = "Open" });

var one = await connection.QuerySingleOrDefaultAsync<Offering>(
    "SELECT * FROM offerings WHERE slug = @slug",
    new { slug = "seed-round" });
```

Dapper's own type handling applies. Guid columns arrive as strings, so register a handler if Dapper's default mapping does not cover your case:

```
public sealed class GuidTypeHandler : SqlMapper.TypeHandler<Guid>
{
    public override Guid Parse(object value) =>
        value is Guid guid ? guid : Guid.Parse((string)value);

    public override void SetValue(IDbDataParameter parameter, Guid value) =>
```

```
parameter.Value = value.ToString("D");
}

SqlMapper.AddTypeHandler(new GuidTypeHandler());
```

10. Type mapping

CLR type	Column definition	On the wire	Notes
string	TEXT / VARCHAR(n)	string	
bool	BOOLEAN	1 / 0	Reads true/false too
byte...long	INTEGER / BIGINT	number	
float, double	DOUBLE PRECISION	number	
decimal	NUMERIC(38,6)	number	Invariant culture
Guid	VARCHAR(36)	canonical D string	
DateTime	TIMESTAMP	yyyy-MM-dd HH:mm:ss.fffffff	Parsed as UTC
DateTimeOffset	VARCHAR(40)	UTC, same format	Offset not preserved
TimeSpan	VARCHAR(32)	c format	
byte[]	BYTEA	hex / base64	
Enum	integer	number	

Two things to be deliberate about

DateTimeOffset loses its offset. Values are normalised to UTC on write and read back as UTC. That is a deliberate choice: the transport carries no zone, so keeping local time would mean the same row reads as a different instant depending on which machine read it. If the original offset matters to your domain, store it in a separate column.

Column types are inferred, not declared. Rows arrive as JSON with no schema. The reader fixes column names and ordinals from the first row of the first page, and infers each column's CLR type from the first non-null value in that page. A column that is entirely null on the first page types as object.

That schema is then held for the life of the reader. A later page with different keys would otherwise shift what an ordinal means mid-read, and a consumer that has already mapped ordinal 3 to a property has no way to notice.

11. No transactions

The engine exposes no transaction verbs. BeginTransaction throws by default:

```
using var transaction = connection.BeginTransaction();
// NotSupportedException
```

Setting AllowNonTransactional=true returns a scope that satisfies the ADO.NET surface without isolating anything. Even then, Rollback throws — statements were applied as they were issued, and a silent no-op rollback would report a half-written record as cleanly reverted.

```
// AllowNonTransactional=true
using var transaction = connection.BeginTransaction();

// ... statements are already applied as each executes ...

transaction.Commit(); // no-op
transaction.Rollback(); // NotSupportedException
```

What to do instead

Order writes so a partial failure is recoverable. Write the dependent record first and the record that makes it visible last, so a failure leaves an orphan rather than a dangling reference.

```
// Investor and allocation first. If the subscription write fails, they are
// orphaned - findable and cleanable - rather than referenced by a subscription
// that has no allocation.
await db.InsertAsync(investor, token: ct);
await db.InsertAsync(allocation, token: ct);
await db.InsertAsync(subscription, token: ct);
```

Compensate explicitly.

```
var written = new List<Guid>();

try
{
    await db.InsertAsync(investor, token: ct);
    written.Add(investor.Id);

    await db.InsertAsync(allocation, token: ct);
    written.Add(allocation.Id);
}
catch
{
    foreach (var id in written)
        await TryDeleteAsync(id, ct);

    throw;
}
```

Or keep multi-table writes on a database that has transactions. For a workflow where partial application is genuinely unacceptable — a securities subscription spanning investor, allocation and table records — that is the honest answer.

12. No DDL, no constraints

The engine is schema-on-write. A table comes into being when the first row is written, named for the record type supplied to the save.

Statement	Behaviour
CREATE TABLE	Generated, accepted, dropped
CREATE DATABASE	Creates the table (CreateDatabaseAsync)
ALTER, DROP, TRUNCATE, CREATE INDEX	NotSupportedException

Destructive DDL throws rather than being ignored, because treating a DROP as a no-op would report a table as removed while it still holds every row.

The part that is easy to miss

PRIMARY KEY, NOT NULL, UNIQUE and foreign keys are not enforced.

An [Index(Unique = true)] on Offering.Slug constrains nothing. A second row with the same slug will be accepted, and /invest/seed-round becomes ambiguous.

Enforce it in the application:

```
public async Task CreateAsync(Offering offering, CancellationToken ct = default)
{
```

```

using var db = await dbFactory.OpenDbConnectionAsync(token: ct);

// The engine enforces no unique constraint, so this check is the only one.
// Note it is a read-then-write with no transaction: two concurrent calls can
// both pass. Where that matters, serialise upstream.
if (await db.ExistsAsync<Offering>(o => o.Slug == offering.Slug, token: ct))
    throw new InvalidOperationException($"Slug '{offering.Slug}' is already taken.");

offering.Id = Guid.NewGuid();
offering.CreatedAt = DateTimeOffset.UtcNow;

await db.InsertAsync(offering, token: ct);
}

```

The race window is real and cannot be closed at the data layer. Where uniqueness is load-bearing, serialise through a single writer, a distributed lock, or a deterministic key that makes collisions impossible.

Provisioning a schema

Through the ElephantSqlDB management portal — the Visual Schema Designer, or a Data Migration import from an existing PostgreSQL or SQLite database. Not from application startup.

13. Parameters and injection

The backend accepts a complete statement and binds nothing, so `ElephantParameterBinder` renders values into the SQL text. This is the highest-risk file in the package and the one to review most carefully.

What it does:

- Walks the statement, **skipping string literals and comments**, so a marker inside one is left alone
- Matches names **longest-first**, so `@id` is never substituted inside `@id_2`
- Accepts `@name` and `:name`
- Formats each value from its CLR type under **invariant culture**
- Doubles embedded quotes and strips NUL
- **Throws on any type without an explicit rule**

That last point matters. There is no `ToString` fallback, because that fallback is exactly how a value carrying a quote escapes its literal.

```

// Safe: rendered as a quoted literal with doubled quotes.
command.Parameters.Add(new ElephantParameter("name", "x'; DROP TABLE users; --"));
// → WHERE name = 'x''; DROP TABLE users; --'

// Throws NotSupportedException rather than guessing.
command.Parameters.Add(new ElephantParameter("v", new MyType()));

```

Convert unusual types before assigning `Value`. If you add a type rule, add a test in `ParameterBinderTests` at the same time.

Never concatenate user input into `CommandText`. The binder only protects values passed as parameters.

14. Performance

Opening a connection is expensive. Each `OpenAsync` constructs a client and makes an authentication round trip. There is no connection pool. Open once per unit of work:

```
// Good
using var db = await dbFactory.OpenDbConnectionAsync(token: ct);

foreach (var id in ids)
    results.Add(await db.SingleByIdAsync<Offering>(id, token: ct));

// Bad - one auth round trip per iteration
foreach (var id in ids)
{
    using var db = await dbFactory.OpenDbConnectionAsync(token: ct);
    results.Add(await db.SingleByIdAsync<Offering>(id, token: ct));
}
```

Every command is an HTTP call. Batch with IN clauses rather than looping:

```
var offerings = await db.SelectAsync<Offering>(o => ids.Contains(o.Id), token: ct);
```

Reads are paged. MaxRecords bounds each round trip; the reader fetches the next page when the consumer reads past the end of the current one. A scan of 10,000 rows at the default MaxRecords=100 is 100 HTTP calls. Raise MaxRecords for scan-heavy workloads, or bound the query with LIMIT.

Set PageNumber when you want one page. Left unset, the reader walks forward until the result is exhausted — a query intended as page 1 of a grid becomes a full scan. See [section 7](#).

Use CountAsync rather than COUNT(*). TotalItems arrives with the first page, so a count costs one round trip and a page of rows instead of a second query.

Multi-row inserts are one call per row. InsertAll over 500 records is 500 round trips, with no rollback if one fails partway.

Async is real. The client is HttpClient-based, so ReadAsync and ExecuteReaderAsync do not block a thread-pool thread. Use the async overloads throughout — the sync ones block on the async path.

15. Error handling

```
try
{
    await db.InsertAsync(offering, token: ct);
}
catch (ElephantException ex)
{
    // Carries the client's AuthStatus.
    logger.LogError(ex, "Insert failed with {Status}.", ex.Status);

    if (ex.Status is AuthStatus.TokenInvalid or AuthStatus.Unauthenticated)
        await RefreshTokenAsync(ct);

    throw;
}
catch (NotSupportedException ex)
{
    // Unsupported DDL, a transaction, or a parameter type with no literal rule.
    logger.LogError(ex, "Unsupported operation.");
    throw;
}
```

Exception	Cause
InvalidOperationException	Connection not open, CommandText empty, required keyword missing
ElephantException	Backend returned a non-success AuthStatus
NotSupportedException	DDL, transaction, or unmapped parameter type

Exception	Cause
InvalidCastException	Column value did not convert to the requested type
IndexOutOfRangeException	Unknown column name or ordinal

16. Testing

IElephantExecutor exists so the provider can be tested without a live API. The client's operations are extension methods on IElephantSqlDBAPIService and cannot be substituted, so without the seam nothing above it would be testable.

```
[Fact]
public async Task Parses_guid_from_string_column()
{
    var id = Guid.Parse("0192f9a0-1111-7222-8333-444455556666");

    var fake = new FakeElephantExecutor().EnqueuePage(
        FakeElephantExecutor.Row(("id", id.ToString("D"))));

    await using var connection = await fake.OpenConnectionAsync();
    using var command = connection.CreateCommand();
    command.CommandText = "SELECT id FROM offerings";

    using var reader = await command.ExecuteReaderAsync();
    Assert.True(await reader.ReadAsync());

    Assert.Equal(id, reader.GetGuid(0));
}
```

FakeElephantExecutor records every statement after binding, which is how the binder is asserted end to end:

```
Assert.Equal(
    "SELECT id FROM offerings WHERE slug = 'seed-round'",
    fake.Statements[0]);
```

Use EnqueuePartialPage to force the reader's paging path — a small fixture never takes it otherwise. It also sets TotalItems independently of the row count, which is how the pagination assertions are written:

```
var fake = new FakeElephantExecutor()
    .EnqueuePartialPage(1, 25, 412, FakeElephantExecutor.Row(("id", "1")));

// ... reader.TotalItems == 412, reader.Pagination.TotalPages == 17
```

In your own application, inject a custom executor factory:

```
var connection = new ElephantConnection(connectionString, _ => myFakeExecutor);
```

17. Verification

Against a real table, in this order. Each depends on the one before.

1. **DoesTableExist** — exercises connection, command, binder and reader in a single call, and is the only schema operation the documentation confirms.
2. **Guid round trip** — insert and read back a primary key.
3. **DateTimeOffset round trip** — confirm UTC normalisation survives.
4. **Value fidelity through the parser** — parameters are rendered into SQL by the binder, then extracted back out by ElephantSqlParser into InsertValuesRows. Whether a value containing an apostrophe or a comma survives that round trip intact is the open question. Test a string with an apostrophe and a Guid.

5. **Multi-row insert** — confirm every row lands.
6. **Paging** — a result set larger than MaxRecords.

Steps 2 and 4 are the ones most likely to surface a problem.

18. Migrating from PostgreSQL

One line changes:

```
// Before
new OrmLiteConnectionFactory(connectionString, PostgreSQLDialect.Provider);

// After
new OrmLiteConnectionFactory(connectionString, ElephantOrmLiteDialectProvider.Instance);
```

What else needs attention:

Concern	Action
[Schema] attributes	Ignored. Leave them or remove them.
search_path pinning	No equivalent. The table is the schema.
Unique indexes	Not enforced. Add application-level checks.
Foreign keys	Not enforced.
Transactions	Removed. Reorder writes or compensate.
SchemaBootstrapper / DDL migrations	Move to a Data Migration import.
Npgsql-specific code	NpgsqlConnectionStringBuilder, set_config, DISCARD ALL behaviour, trigger-based audit SQL all need replacing.
[AutoIncrement] keys	Switch to Guid.
DateTimeOffset columns	Offset is no longer preserved.

The schema itself can be imported through the management portal's Data Migration option, which reads an existing PostgreSQL or SQLite database directly.

19. Troubleshooting

InvalidOperationException: ConnectionStrings:... is not set — the environment variable did not bind. Check the key name and, in Kubernetes, that the secret exists in the right namespace.

ElephantException with TokenInvalid — the bearer token expired. Open validates credentials, so this surfaces at connection time.

ElephantException with CredentialsNotFound or NoTenantDatabaseFound — the account or the table does not exist. Provision it, or set `AutoRegisterUser` / `AutoCreateTable` for a development environment. See [section 4](#).

An empty table appeared — `AutoCreateTable` was on and Database held a typo. Turn it off outside development.

NotSupportedException on db.Save() — OrmLite's upsert path reached for a transaction. Use explicit `InsertAsync/UpdateAsync`, or set `AllowNonTransactional=true` knowing it buys no atomicity.

InvalidCastException on a column that is sometimes null — the column was entirely null on the first page, so it typed as object. Check `IsDBNull` first, or order the query so a non-null value appears early.

A SELECT returns fewer rows than expected — `MaxRecords` bounds each page and the reader pages forward. If you are reading through something that does not walk the reader to the end, you see one page. Check `reader.Pagination.TotalItems` against what you received.

A paged query is making far more calls than expected — `PageNumber` was not set, so the reader walked every page. Set it, or use `QueryPageAsync`.

`GetPagination()` returns null — the command or reader did not come from this provider. That happens behind an abstraction that wraps or proxies the ADO.NET types.

Duplicate rows where a unique index exists — the index is not a constraint. See [section 12](#).

A row written after `ChangeDatabase` cannot be read back — the read is running against a different database from the write. Check `connection.Database` and any `command.DatabaseName` on the reading command, and check that the connection was not re-created (a new connection starts on the `Database` keyword). See [Choosing the database](#).

Values read back wrapped in quotes, NULL as text, or ' ' in place of ' — the rows were written through the client's `InsertSQLRecordsAsync` (directly, or by an older build of this provider), which stores SQL literals verbatim. Rows written by this provider are decoded; earlier rows need repairing.

Three upstream client issues this package works around

Worth raising with whoever owns `ElephantSqlDBCClient.Core`, since the workarounds should not be permanent:

1. `InsertSQLRecordsAsync` reads only `InsertValuesRows[0]` and discards the rest — a multi-row `INSERT` writes one row. The provider bypasses it and calls `SaveToDatabaseWithNoResponse` once per row.
2. `InsertSQLRecordsAsync` returns `AuthStatus.Success` unconditionally; the save result is never inspected and every exception is folded into `SQLStatementInvalid`. The provider inspects the boolean directly.
3. `InsertSQLRecordsAsync` stores SQL literals verbatim — `'abc'` keeps its quotes, `NULL` is stored as the text `NULL`, and `' '` is not collapsed to `'`. The provider decodes each token (`DecodeLiteral`, the inverse of `ElephantParameter.FormatLiteral`) before saving. Code calling `InsertSQLRecordsAsync` directly is affected.

`NewRecord` in `ElephantSqlDbExecutor` also duplicates the client's private `AddCommonInterfaceProperties`. If a key is added there and missed here, the server accepts a record that is quietly incomplete — worth making that method internally visible, or pinning it with a test.