Pitfalls of Relational DB access: rethinking .NET micro-ORMs

Stan Drapkin

May 2019

Who am I?

- Stan Drapkin – sdrapkin@sdprime.com
- CTO of IT firm (cybersecurity & regulatory compliance)
- OSS library author (github.com/sdrapkin)
  - **TinyORM** – *.NET micro ORM done right*
  - **Inferno** – *.NET crypto done right*
- Book author
  - “**SecurityDriven .NET**” (2014)
  - “**Application Security in .NET, Succinctly**” (2017)

Agenda

DB-access myths

Optimizations

Myth 1

ADO.NET: just add Async.

ADO.NET – idiomatic code

```
conn.Open(); // step-1
var reader = comm.ExecuteReader(); // step-2
do
{
        while (reader.Read()) // step-3
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data); // ie. using the data; step-4
        }
}
while (reader.NextResult()); // step-5
```

async programming – docs.microsoft.com

You can avoid performance bottlenecks *and enhance application responsiveness with async…*

…but then…

using async will have ***no noticeable benefits and even could be detrimental****.*

"Use tests, profiling and ***common sense*** *…"*

ADO.NET – let’s async it with common sense

```
conn.Open(); // #1
var reader = comm.ExecuteReader(); // #2
do
{
	while (reader.Read()) // #3
	{
		var data = new object[reader.FieldCount];
		reader.GetValues(data);
		WriteLine(data); // ie. using the data; #4
	}
}
while (reader.NextResult()); // #5
```

ADO.NET – async'ifying… via common sense

```
conn.Open(); // #1                                   Async 1
var reader = comm.ExecuteReader(); // #2             Async 2
do
{
        while (reader.Read()) // #3                  Async 3
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data); // ie. using the data; #4
        }
}
while (reader.NextResult()); // #5                   Async 4
```

ADO.NET – async – done.

```
await conn.OpenAsync(); // #1
var reader = await comm.ExecuteReaderAsync(); // #2
do
{
	while (await reader.ReadAsync()) // #3
	{
		var data = new object[reader.FieldCount];
		reader.GetValues(data);
		WriteLine(data); // ie. using the data; #4
	}
}
while (await reader.NextResultAsync()); // #5
```

ADO.NET – async – can we improve?

```
await conn.OpenAsync();                                    Task t1
var reader = await comm.ExecuteReaderAsync();              Task t2
do
{
        while (await reader.ReadAsync())                   Task t3
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data);
        }
}
while (await reader.NextResultAsync());                    Task t4
```

ADO.NET – async – can we improve?

```
await conn.OpenAsync();                          t1.IsComplete?
var reader = await comm.ExecuteReaderAsync();    t2
do
{
        while (await reader.ReadAsync())         t3
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data);
        }
}
while (await reader.NextResultAsync());          t4
```

ADO.NET – async – can we improve?

```
await conn.OpenAsync();                                   Completed.
var reader = await comm.ExecuteReaderAsync();                     t2
do
{
        while (await reader.ReadAsync())                          t3
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data);
        }
}
while (await reader.NextResultAsync());                           t4
```

ADO.NET – async – can we improve?

```
await conn.OpenAsync();                                      Completed.
var reader = await comm.ExecuteReaderAsync();t2.IsComplete?
do
{
        while (await reader.ReadAsync())                           t3
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data);
        }
}
while (await reader.NextResultAsync());                            t4
```

ADO.NET – async – can we improve?

```
await conn.OpenAsync();                                   Completed.
var reader = await comm.ExecuteReaderAsync();            Incomplete.
do
{
        while (await reader.ReadAsync())                          t3
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data);
        }
}
while (await reader.NextResultAsync());                           t4
```

ADO.NET – async – can we improve?

```
await conn.OpenAsync();                                  Completed.
var reader = await comm.ExecuteReaderAsync();           Incomplete.
do
{
        while (await reader.ReadAsync())              t3.IsComplete?
        {
                var data = new object[reader.FieldCount];
                reader.GetValues(data);
                WriteLine(data);
        }
}
while (await reader.NextResultAsync());               t4.IsComplete?
```

ADO.NET – async – can we improve?

```
await conn.OpenAsync();                          Completed.
var reader = await comm.ExecuteReaderAsync();    Incomplete.
do
{
    while (await reader.ReadAsync())             Completed.
    {
        var data = new object[reader.FieldCount];
        reader.GetValues(data);
        WriteLine(data);
    }
}
while (await reader.NextResultAsync());          Completed.
```

ADO.NET – async – improved ✓

```
conn.Open();
var reader = await comm.ExecuteReaderAsync();
do
{
	while (reader.Read())
	{
		var data = new object[reader.FieldCount];
		reader.GetValues(data);
		WriteLine(data);
	}
}
while (reader.NextResult);
```

Async optimizations – summary:

- Mostly **completed** tasks:
  - state-machine stack allocation (40+ bytes)
  - state capture & field assignment (depends on closures)
  - GetAwaiter() call
  - IsCompleted call

  Better off with Sync-version.

- Mostly **incomplete** tasks:

  Async-version might be preferred.

Myth 2

DbConnections must be there.

Connections are an anti-pattern

Most micro-ORMs are connection-oriented

Dapper is just IDbConnection extensions

We continue to:

- Create, Open, Close, Track, and Dispose connections
- Pass connections through layers and contexts

Why do we keep doing this ?

Connections are an anti-pattern

We’ve been **conditioned** to treat connections as a norm

- 17 years of ADO.NET patterns hammered into us

Connections are a **low-level** implementation detail

- Must be hidden and transparent
- Like async State-Machine, should be done by tooling

Stop managing connections

Connections are an anti-pattern

High-level db concept is a **transaction** – not connection.

Connections should be:

- Auto-created & auto-disposed, as needed
- Auto-enlisted in transactions, as needed

TinyORM is one example of connection-free micro-ORM.

Myth 3

Must. Have. POCOs.

...my precious...

Data Transfer Object (DTO)

```
while (reader.Read())
{
    object[] data = new object[reader.FieldCount];
    reader.GetValues(data);
    WriteLine(data);
}
```

data can be stored inside a simple container: DTO

Concepts – **POCO** vs **DTO**

```
class Cat
{
  Guid Id;
  string Name;
  int Age;
}
// POCO
```

```
class DTO: IDynamicMetaObjectProvider
{
  object[] Data;

  RSSchema schema; // field name info
}
// similar to Dapper DTO
```

Concepts – **POCO** vs **DTO** – how to use

```
// POCO
List<Cat> cats;
Cat c = cats[0];


Guid id = c.Id;
```

```
// DTO
List<DTO> cats;
DTO c = cats[0];


Guid id1 = c["Id"];          // string indexer
Guid id2 = c[0];             // int indexer

dynamic d = cats[0];
Guid id3 = d.Id;             // dynamic call
```

POCO vs **DTO** – summary

POCOs are nice but costly – not a must-have.
DTOs can work just as well.

Why & when to prefer DTOs to POCOs?

…will be answered later.

Myth 4

micro-ORMs need lots of APIs.

Dapper API: 20+ methods

Close
Execute
ExecuteAsync
ExecuteReaderAsync
ExecuteScalar
ExecuteScalarAsync
Open
OpenAsync
Query
QueryAsync

QueryFirst
QueryFirstAsync
QueryFirstOrDefault
QueryFirstOrDefaultAsync
QueryMultiple
QueryMultipleAsync
QuerySingle
QuerySingleAsync
QuerySingleOrDefault
QuerySingleOrDefaultAsync

EF.Core API: 20+ methods

Add
AddAsync
AddRange
AddRangeAsync
Attach
AttachRange
Entry
Find
FindAsync
Remove
RemoveRange
SaveChanges
SaveChangesAsync
Set
Update
UpdateRange
FromSql
ExecuteSqlCommand
OnConfiguring, OnModelCreating
LINQ querying via IQueryable

ORMLite API: 20…

ColumnAsync
ColumnDistinctAsync
ColumnDistinctFmtAsync
ColumnFmtAsync
DictionaryAsync
DictionaryFmtAsync
ExecuteNonQueryAsync
ExistsAsync
ExistsFmtAsync
LoadReferencesAsync

LoadSingleByIdAsync
LongScalarAsync
LookupAsync
LookupFmtAsync
ScalarAsync
ScalarFmtAsync
SelectAsync
SelectByIdsAsync
SelectFmtAsync
SelectNonDefaultsAsync

ORMLite API: 40…

SingleAsync
SingleByIdAsync
SingleFmtAsync
SingleWhereAsync
SqlColumnAsync
SqlListAsync
SqlProcedureAsync
SqlProcedureFmtAsync
SqlScalarAsync
WhereAsync

DeleteAll
DeleteAllAsync
DeleteByIdAsync
DeleteByIdsAsync
DeleteFmtAsync
DeleteNonDefaultsAsync
ExecuteProcedureAsync
InsertAllAsync
InsertAsync
SaveAllAsync

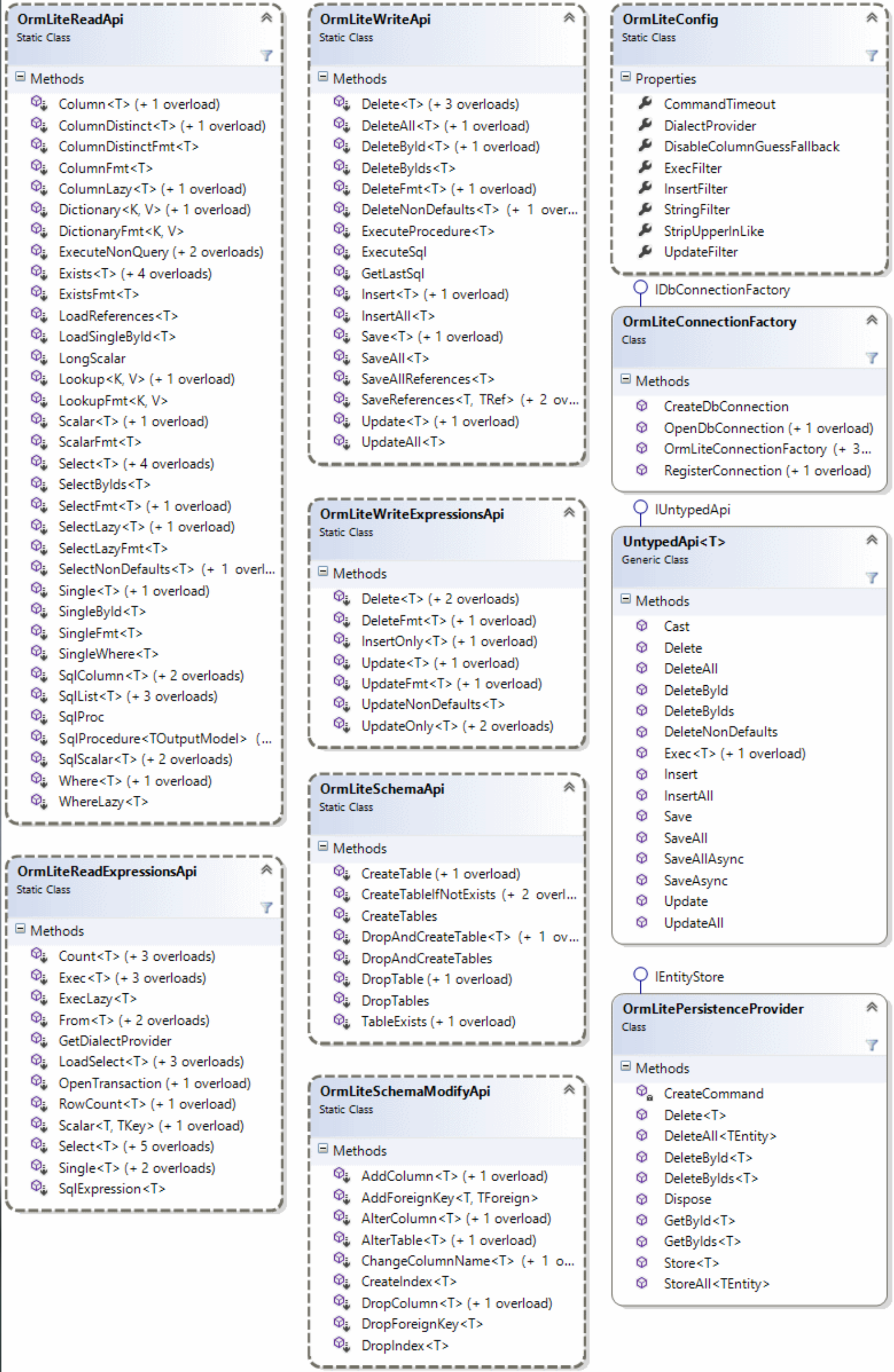
ORMLite API: 77… + 60 = 137. Madness.

SaveAllReferencesAsync
SaveAsync
SaveReferencesAsync
UpdateAllAsync
UpdateAsync
DeleteAsync
DeleteFmtAsync
InsertOnlyAsync
UpdateFmtAsync
UpdateNonDefaultsAsync

UpdateOnlyAsync
CountAsync
LoadSelectAsync
RowCountAsync
ScalarAsync
SelectAsync
SingleAsync

...that’s just Async
60+ more sync methods.


LEAD
DEV


OrmLiteReadApi
Static Class
Methods
Column<T> (+ 1 overload)
ColumnDistinct<T> (+ 1 overload)
ColumnDistinctFmt<T>
ColumnFmt<T>
ColumnLazy<T> (+ 1 overload)
Dictionary<K, V> (+ 1 overload)
DictionaryFmt<K, V>
ExecuteNonQuery (+ 2 overloads)
Exists<T> (+ 4 overloads)
ExistsFmt<T>
LoadReferences<T>
LoadSingleById<T>
LongScalar
Lookup<K, V> (+ 1 overload)
LookupFmt<K, V>
Scalar<T> (+ 1 overload)
ScalarFmt<T>
Select<T> (+ 4 overloads)
SelectByIds<T>
SelectFmt<T> (+ 1 overload)
SelectLazy<T> (+ 1 overload)
SelectLazyFmt<T>
SelectNonDefaults<T> (+ 1 overl...
Single<T> (+ 1 overload)
SingleById<T>
SingleFmt<T>
SingleWhere<T>
SqlColumn<T> (+ 2 overloads)
SqlList<T> (+ 3 overloads)
SqlProc
SqlProcedure<TOutputModel> (...
SqlScalar<T> (+ 2 overloads)
Where<T> (+ 1 overload)
WhereLazy<T>
OrmLiteReadExpressionsApi
Static Class
Methods
Count<T> (+ 3 overloads)
Exec<T> (+ 3 overloads)
ExecLazy<T>
From<T> (+ 2 overloads)
GetDialectProvider
LoadSelect<T> (+ 3 overloads)
OpenTransaction (+ 1 overload)
RowCount<T> (+ 1 overload)
Scalar<T, TKey> (+ 1 overload)
Select<T> (+ 5 overloads)
Single<T> (+ 2 overloads)
SqlExpression<T>
OrmLiteWriteApi
Static Class
Methods
Delete<T> (+ 3 overloads)
DeleteAll<T> (+ 1 overload)
DeleteById<T> (+ 1 overload)
DeleteByIds<T>
DeleteFmt<T> (+ 1 overload)
DeleteNonDefaults<T> (+ 1 over...
ExecuteProcedure<T>
ExecuteSql
GetLastSql
Insert<T> (+ 1 overload)
InsertAll<T>
Save<T> (+ 1 overload)
SaveAll<T>
SaveAllReferences<T>
SaveReferences<T, TRef> (+ 2 ov...
Update<T> (+ 1 overload)
UpdateAll<T>
OrmLiteWriteExpressionsApi
Static Class
Methods
Delete<T> (+ 2 overloads)
DeleteFmt<T> (+ 1 overload)
InsertOnly<T> (+ 1 overload)
Update<T> (+ 1 overload)
UpdateFmt<T> (+ 1 overload)
UpdateNonDefaults<T>
UpdateOnly<T> (+ 2 overloads)
OrmLiteSchemaApi
Static Class
Methods
CreateTable (+ 1 overload)
CreateTableIfNotExists (+ 2 overl...
CreateTables
DropAndCreateTable<T> (+ 1 ov...
DropAndCreateTables
DropTable (+ 1 overload)
DropTables
TableExists (+ 1 overload)
OrmLiteSchemaModifyApi
Static Class
Methods
AddColumn<T> (+ 1 overload)
AddForeignKey<T, TForeign>
AlterColumn<T> (+ 1 overload)
AlterTable<T> (+ 1 overload)
ChangeColumnName<T> (+ 1 o...
CreateIndex<T>
DropColumn<T> (+ 1 overload)
DropForeignKey<T>
DropIndex<T>
OrmLiteConfig
Static Class
Properties
CommandTimeout
DialectProvider
DisableColumnGuessFallback
ExecFilter
InsertFilter
StringFilter
StripUpperInLike
UpdateFilter
IDbConnectionFactory
OrmLiteConnectionFactory
Class
Methods
CreateDbConnection
OpenDbConnection (+ 1 overload)
OrmLiteConnectionFactory (+ 3...
RegisterConnection (+ 1 overload)
IUntypedApi
UntypedApi<T>
Generic Class
Methods
Cast
Delete
DeleteAll
DeleteById
DeleteByIds
DeleteNonDefaults
Exec<T> (+ 1 overload)
Insert
InsertAll
Save
SaveAll
SaveAllAsync
SaveAsync
Update
UpdateAll
IEntityStore
OrmLitePersistenceProvider
Class
Methods
CreateCommand
Delete<T>
DeleteAll<TEntity>
DeleteById<T>
DeleteByIds<T>
Dispose
GetById<T>
GetByIds<T>
Store<T>
StoreAll<TEntity>

TinyORM API: 2 methods

QueryAsync	1 result set
QueryMultipleAsync	multiple result sets

Task<List<RowStore>>	QueryAsync
Task<List<List<RowStore>>>	QueryMultipleAsync

Myth 5

Dapper is **easy to use**.

Simple T-SQL

```
var sql = @"
SELECT

  @name                          AS [Name],
  transaction_isolation_level AS [ISOLATION_LEVEL]

FROM sys.dm_exec_sessions
WHERE session_id = @@SPID";
```

Dapper – simplest query

```
await conn.QueryAsync(sql, new { @name = “Hector” });
```

So far, so good.

Name	ISOLATION_LEVEL
Hector	2 (ReadCommitted)

Dapper – simplest query – in a transaction

```
using (var ts = new TransactionScope())
{
  await conn.QueryAsync(sql, new { @name = “Hector” });
  ts.Complete();
}
```

InvalidOperationException

A TransactionScope must be disposed
on the same thread that it was created.

Dapper – simplest query – in a transaction

```
using (var ts = new TransactionScope(
        TransactionScopeAsyncFlowOption.Enabled))
{
  await conn.QueryAsync(sql, new { @name = “Hector” });
  ts.Complete();
}
```

Name	ISOLATION_LEVEL
Hector	4 (Serializable)

Dapper – summary

Dapper is ADO.NET with less code.

All ADO.NET problems are still there.

Same old API paradigms and low-level concepts.

But at least Dapper is fast, right?

Myth 6

Dapper is **fast**.

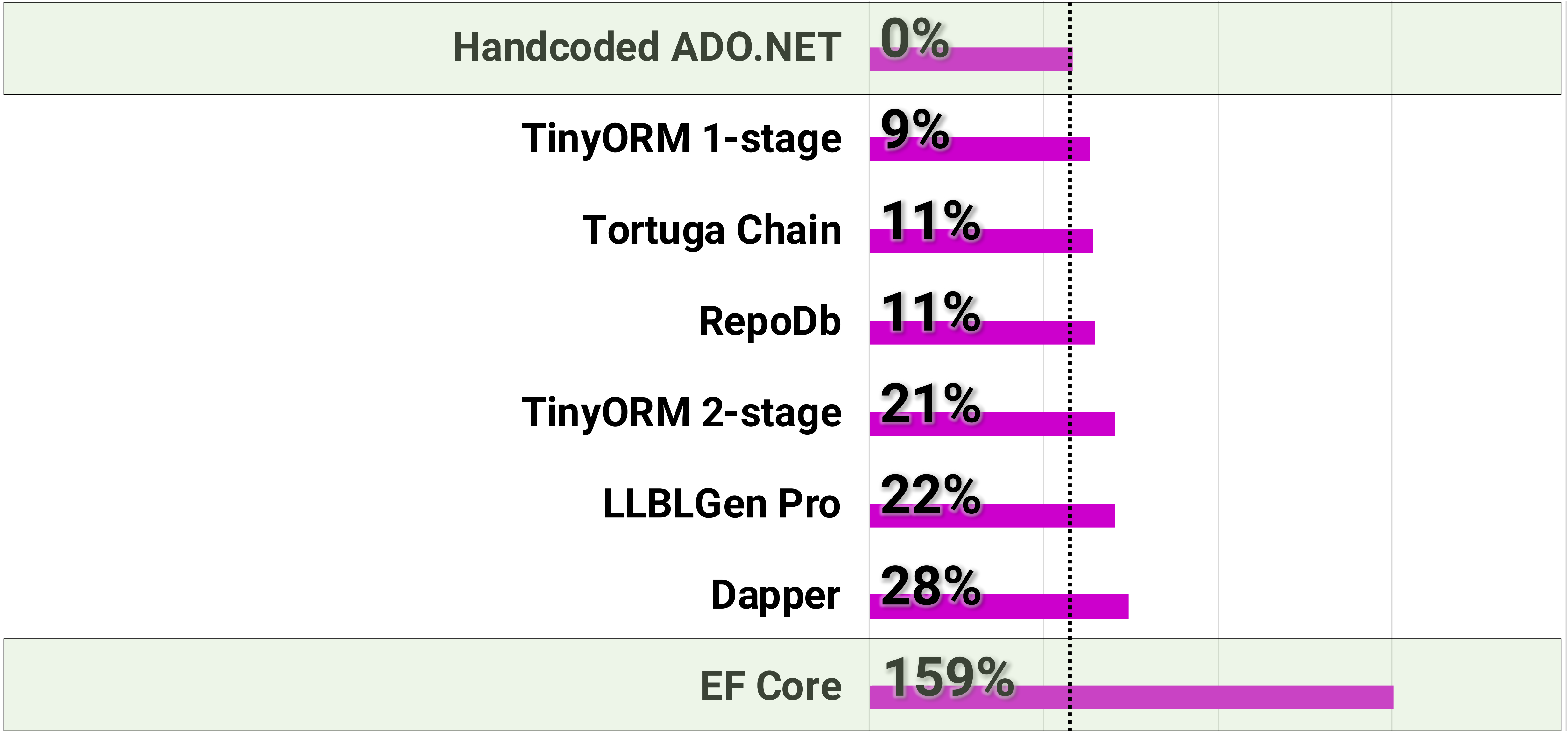
micro-ORM bench – RawDataAccessBencher

- github.com/FransBouma/RawDataAccessBencher
- Mature (from 2013)
- 14+ different micro-ORMs benched
- Not very precise or accurate, but ok for comparisons
- Authored by Frans Bouma (sells LLBLGen Pro)

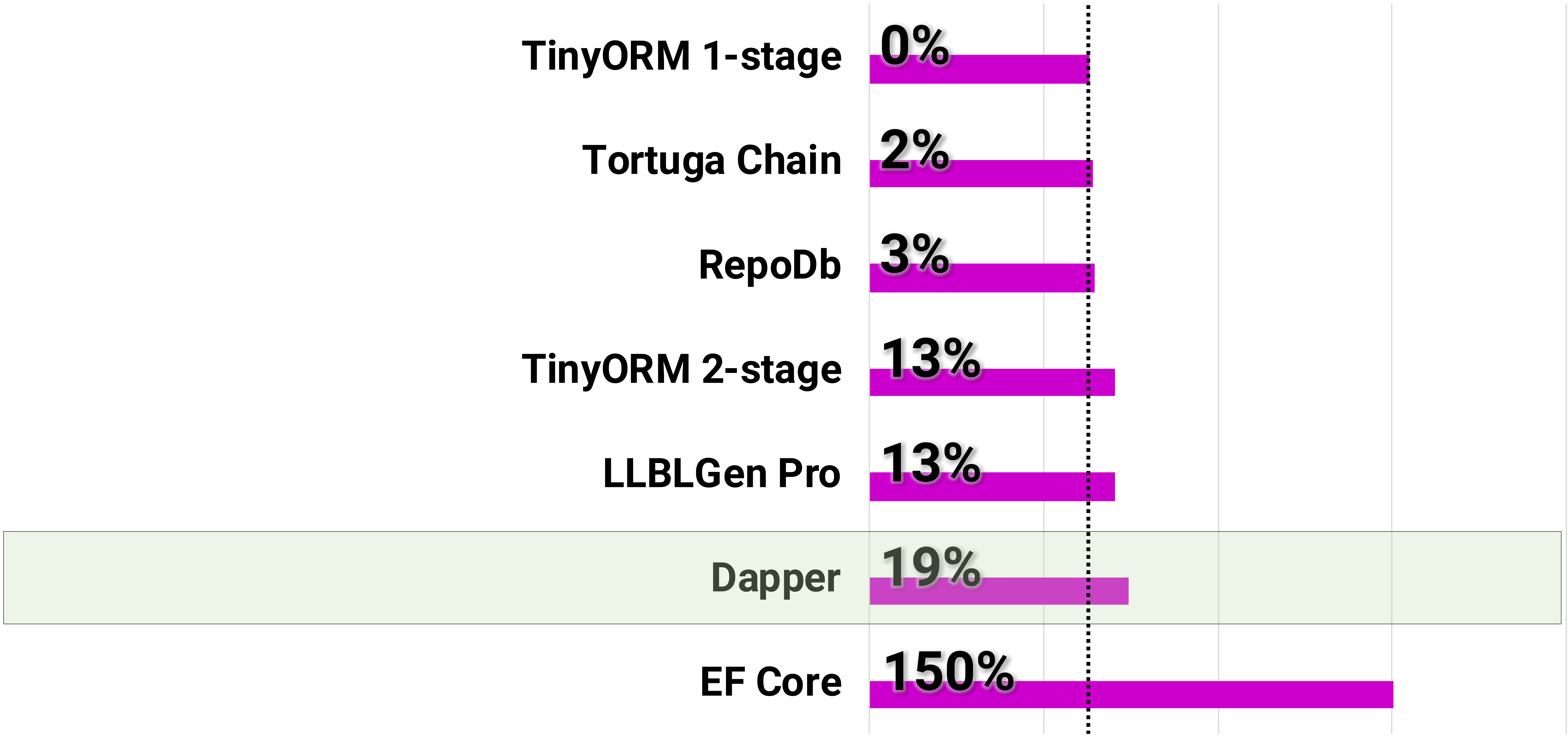
I tested on .NET 4.7.2 x64; Windows-10-latest

Latest versions of all tested micro-ORMs used

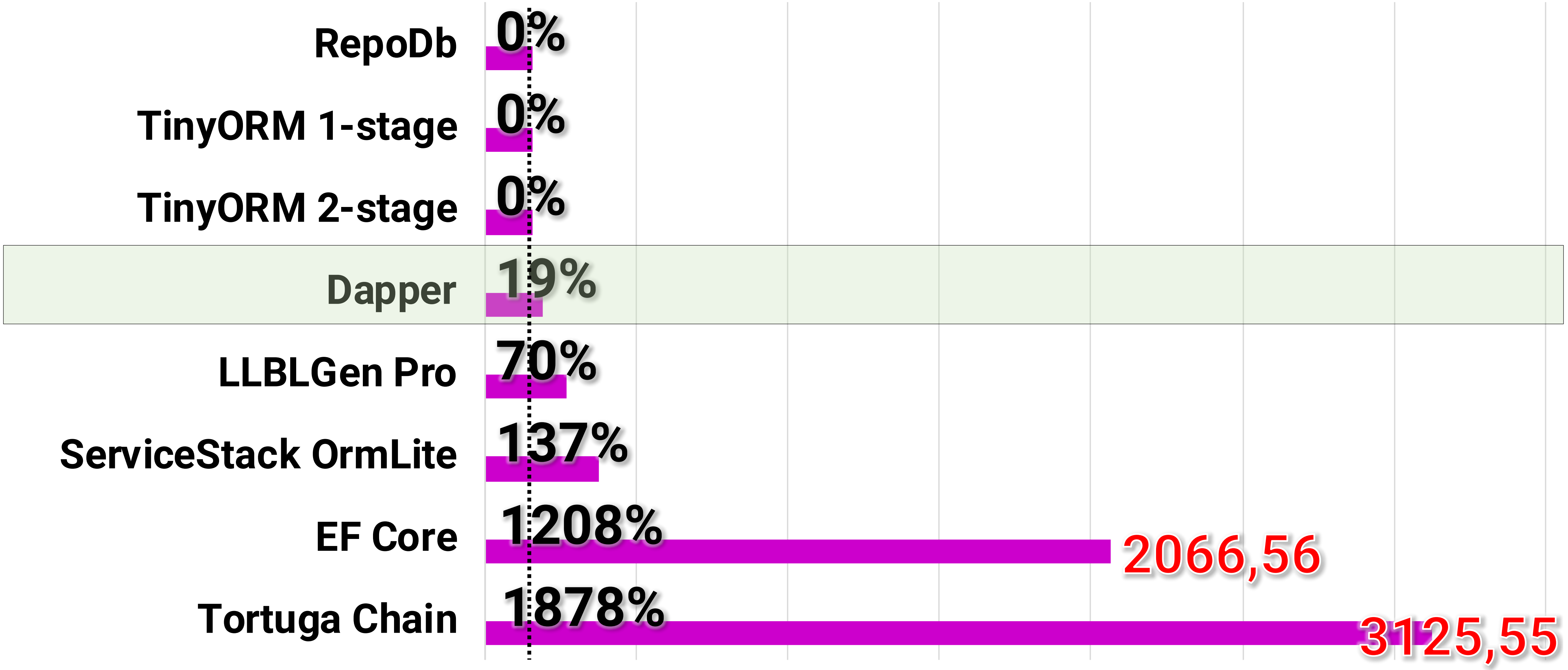
Time & overhead: **31,465** rows to POCOs



Same, but without handcoded ADO.NET



1st-query timings (ms), overhead



Dapper – summary

Dapper can be ~20% slower vs. the fastest micro-ORM.

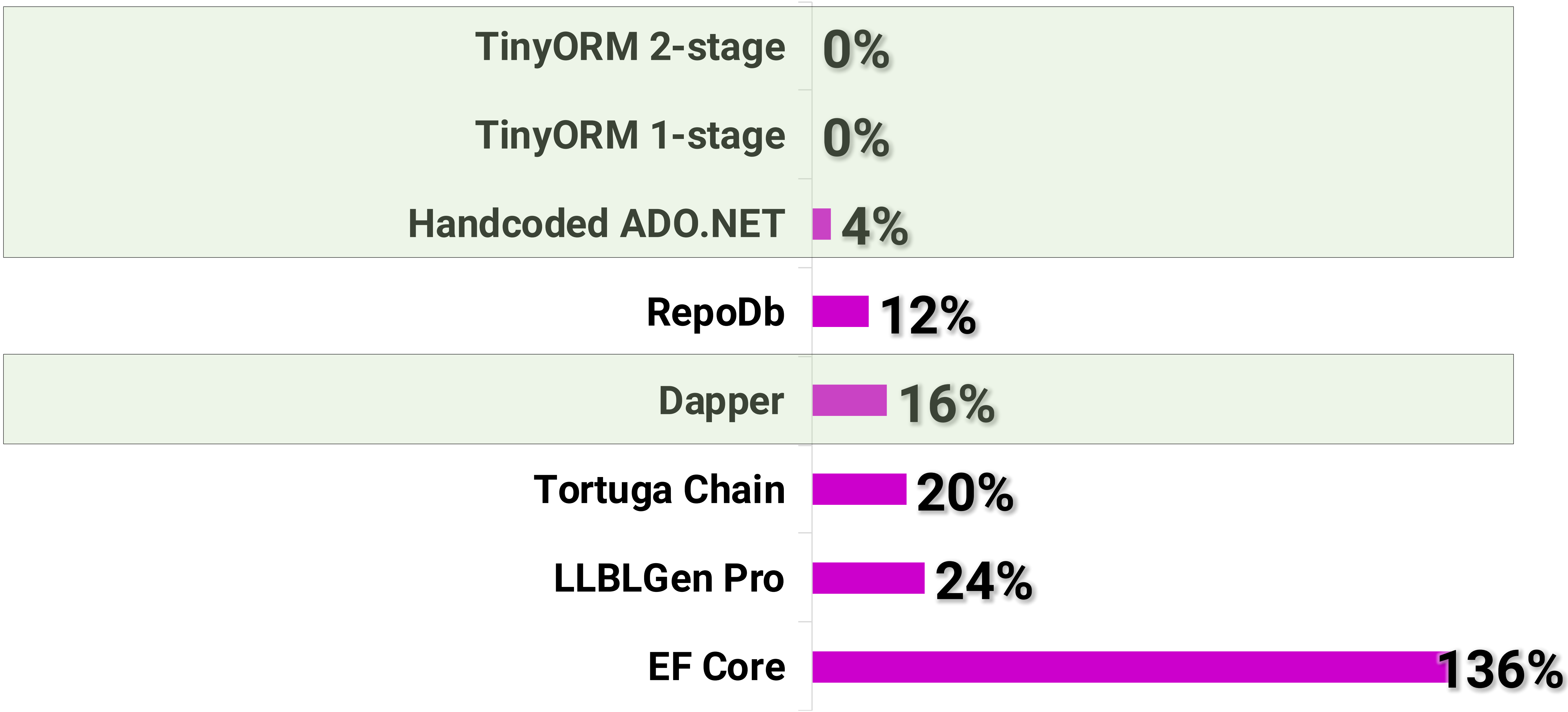
TinyORM: **1-stage** & **2-stage** – what’s that?

… will be covered later.

Myth 7

You can’t beat ADO.NET.

Benchmarks – single-row; normalized



4% faster than ADO.NET. WTF... How ?

```
var conn = new SqlConnection(connString);
```

ADO.NET Connection Pooling

Use your own transaction-aware connection cache

- Don’t defer to ADO.NET connection-pool
- ex. TinyORM/ConnectionCache.cs on GitHub

Result: faster connection setup & teardown

Myth 8

Only **one** micro-ORM approach.

At least 3 distinct useful approaches:

1-stage

½-stage

2-stage

1-stage vs ½-stage

1-stage

1. Connect
2. Send Query
3. Get Data Reader
4. Loop → List<POCO>
5. Disconnect
6. Return List<POCO>

½-stage

1. Connect
2. Send Query
3. Get Data Reader
4. Loop → List<DTO>
5. Disconnect
6. Return List<DTO>

1-stage vs 2-stage

1-stage

1. Connect
2. Send Query
3. Get Data Reader
4. Loop → List<POCO>
5. Disconnect
6. Return List<POCO>

2-stage

1. Connect
2. Send Query
3. Get Data Reader
4. Loop → List<DTO>
5. Disconnect
6. List<DTO> → List<POCO>
7. Return List<POCO>

micro-ORM – stages comparison

	POCO 1	DTO ½	POCO 2
Client performance	yellow	green	yellow
Memory efficiency	yellow	green	yellow
Server efficiency	red	green	green
Multiple disconnected resultset API	yellow	yellow	green
POCOs	green	yellow	green

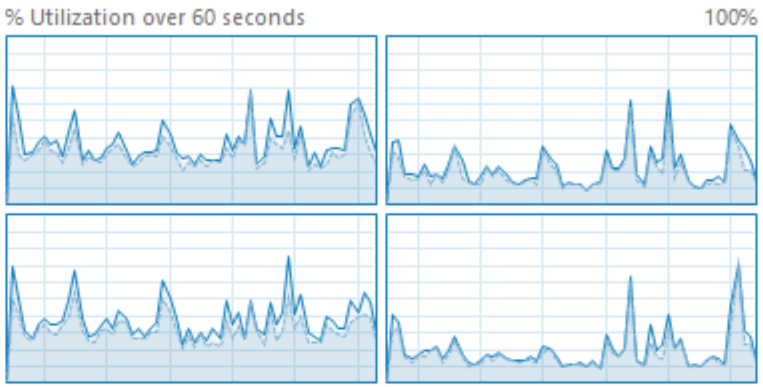
When to prefer DTOs to POCOs?

Give it back!

Async:

Give back my threads!

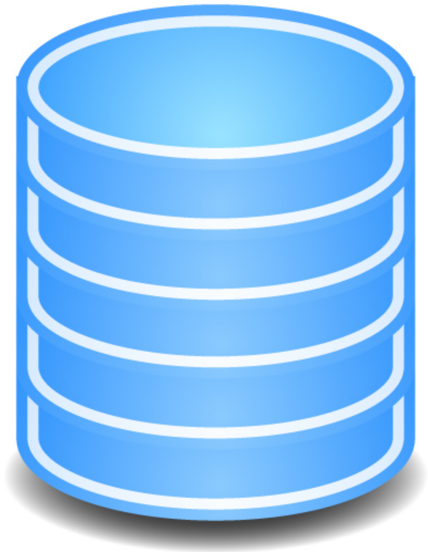
I’m not waiting for your silly I/O.



2-Stage fast-disconnect:

Give back my DB resources!

I’m not waiting for your silly Materialization.



Optimization tricks:

faster DTO

Optimization tricks – Dapper DTO

```
class Dapper_DTO
{
  object[] data;

  RSSchema schema;
}
// later:
// List<Dapper_DTO>
```

```
class RSSchema
{// ResultSetSchema
  string[] fieldNames;

  Dictionary<string, int>
    fieldNameLookup;
}
```

Optimization tricks – Dapper vs TinyORM DTO

```
class Dapper_DTO
{
  object[] data;

  RSSchema schema;
}
```

```
struct TinyORM_DTO
{
  object[] data;
}
```

How is this any better ?

Where did schema go ?

DTO memory layout – x64

`Dapper_DTO object layout:`

`Size: 32 bytes.`

```
|==============================|
| Object Header          (8 b) |
|------------------------------|
| Method Table Ptr       (8 b) |
|==============================|
| 0-7: Object[] data     (8 b) |
|------------------------------|
| 8-15: RSSchema schema (8 b)  |
|==============================|
```

`TinyORM_DTO struct layout:`

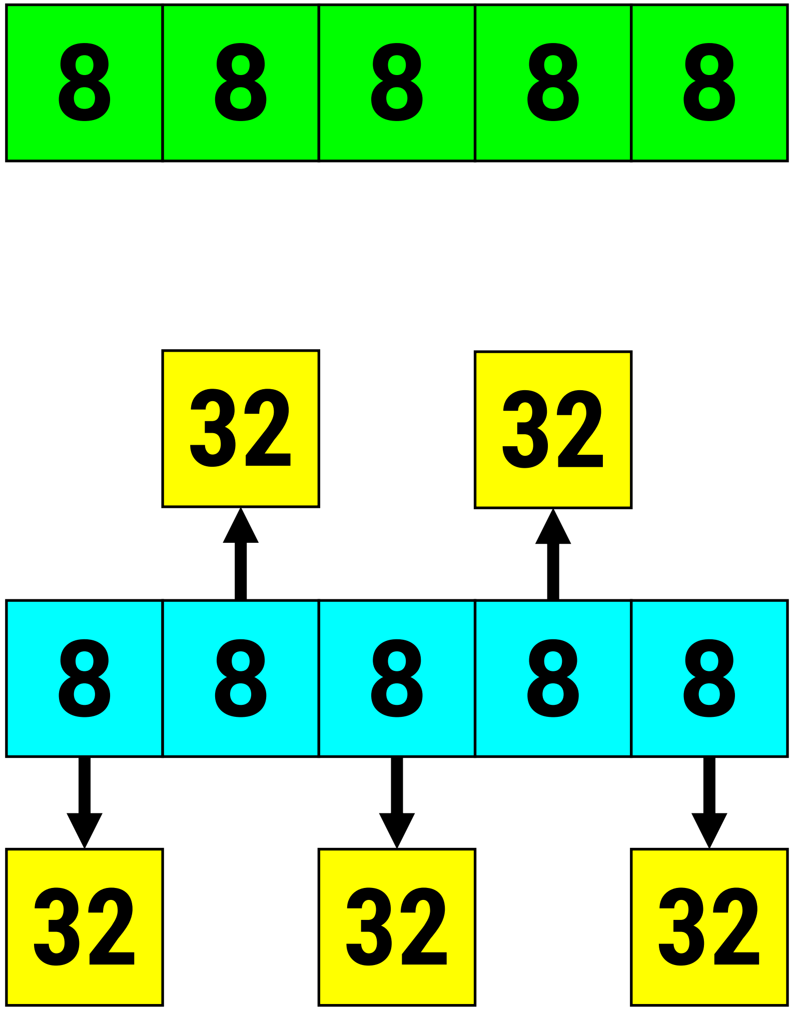
`Size:  8 bytes.`

```
|==============================|
| 0-7: Object[] data     (8 b) |
|==============================|
```

Array of DTO – memory layout – 5x smaller

TinyORM_DTO[5]
1 array of struct
Memory locality ✓

Dapper_DTO[5]
5 extra 32b objects
Fragmentation



Optimization tricks – Dapper vs TinyORM DTO

```
class Dapper_DTO
{
  object[] Data;

  RSSchema schema;
}
```

```
struct TinyORM_DTO
{
  object[] Data;
}
```

✓ How is this any better ?

Where did schema go ??

Recall: row-data extraction

```
while (reader.Read()) // step-3
{
    object[] data = new object[reader.FieldCount];
    reader.GetValues(data);
    WriteLine(data); // ie. using the data; step-4
}
```

Recall: row-data extraction

```
while (reader.Read()) // step-3
{
    object[] data = new object[reader.FieldCount + 1];
    reader.GetValues(data);
    WriteLine(data); // ie. using the data; step-4
}
```

RSSchema is added as the LAST entry in **data**.

Optimization tricks – internals of List<T>

Iteration of List<T> is slow – can we iterate faster?

Optimization tricks – internals of List<T>

```
public class List<T>
{
	private T[] _items; // how can we access?
	…
}
```

Obvious idea: Reflection

- Not fast-enough, even compiled into delegates.
- Reflection overhead reduces any perf. gains.

Optimization tricks – internals of List<T>

```
public class List<T>
{
    private T[] _items;
    ...
}
```

```
public class Tuple<object>
{
    public object Item1;
}
```

1. UNION in-memory layouts
2. Extract private _items via public Item1

Optimization tricks – internals of List<T>

```
[StructLayout(LayoutKind.Explicit)]
struct ListUnion
{
    [FieldOffset(0)]
    public object SomeList; // List<T> input

    [FieldOffset(0)]
    public Tuple<object> ListAccessor; // conversion
}
```

Optimization tricks – internals of List<T>

```
T[] GetList_itemsArray<T>(List<T> list)
{
        return
        (T[])
        new ListUnion { SomeList = list }
        .ListAccessor.Item1;
}
```

Tuple<object>

Optimization tricks – internals of List<T>

```
var list = new List<string>{ "A", "B", "C" };
string[] nativeArray = GetList_itemsArray(list);
foreach (var s in nativeArray) WriteLine(s);
```

A
B
C
null

Thank you for your attention!

Questions?

sdrapkin@sdprime.com
twitter.com/sdrapkin
github.com/sdrapkin

TinyORM – other notable features

- Transaction auto-abort
- POCO change-tracking
- Bulk & batch CUD
- Streaming Reads
- Identity tracking
- Callsite tracking
- SequentialGuids
- Parallel transactions
- Query-building helpers
- TVP support
- Result de-structuring
- .NET Core support*

DB-access myths in .NET:

1. ADO.NET: just add Async, everywhere.
2. DbConnections must be there.
3. Must. Have. POCOs.
4. Micro-ORMs need lots of APIs.
5. Dapper is easy-to-use.
6. Dapper is fast.
7. You can’t be faster than ADO.NET.
8. Only one (1) micro-ORM approach.

9. nvarchar takex 2x space vs. varchar.
10. Clustered GUID PKs are bad – create fragmentation.