Магия Dapper + Oracle

Орлов Юрий

Разработчик C#

03.11.2018

SAR
DOT
NET

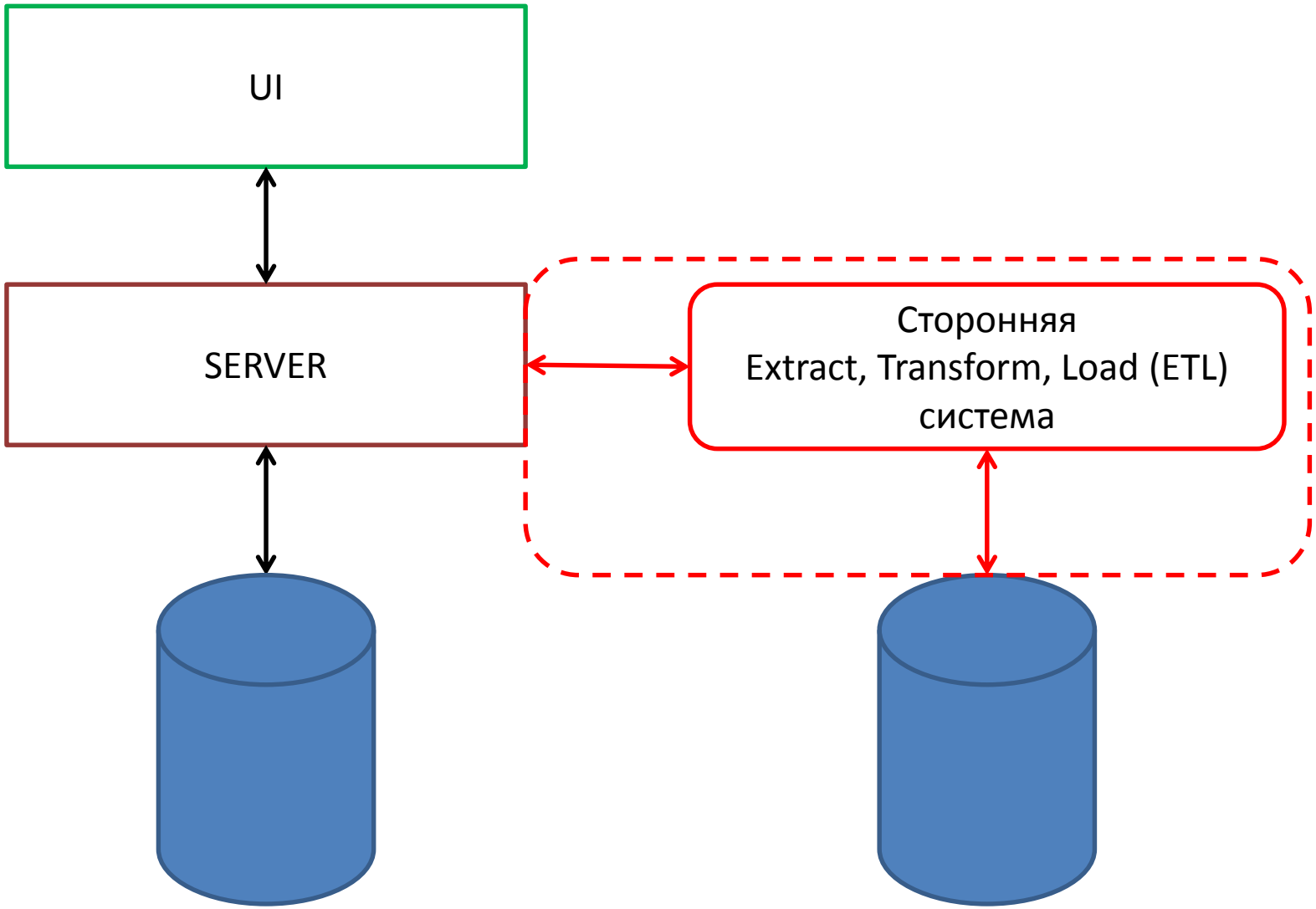
Обо мне

- Разрабатываю серверные приложения на платформе .NET в CUSTIS
- Изучаю балансировку нагрузки с использованием методов для решения NP-трудных задач, пишу статьи в научных изданиях, являюсь инженером-исследователем в ФИЦ ИУ РАН
- Разрабатываю на .NET Core клиент-серверную игру

План

- Пролог
  - История проблемы
  - Что нужно для достижения результата?
- Решение
  - Dapper
  - Ускорение логики на C#
  - Фишки Oracle
- Подведем итоги

ПРОЛОГ



История проблемы

С чего все началось?

Основные требования

- Сложный алгоритм расчета показателей
- Необходимо зачитывать, обрабатывать, и записывать десятки миллионов строк
- Время расчета ограничено несколькими часами

Что нужно для достижения результата?

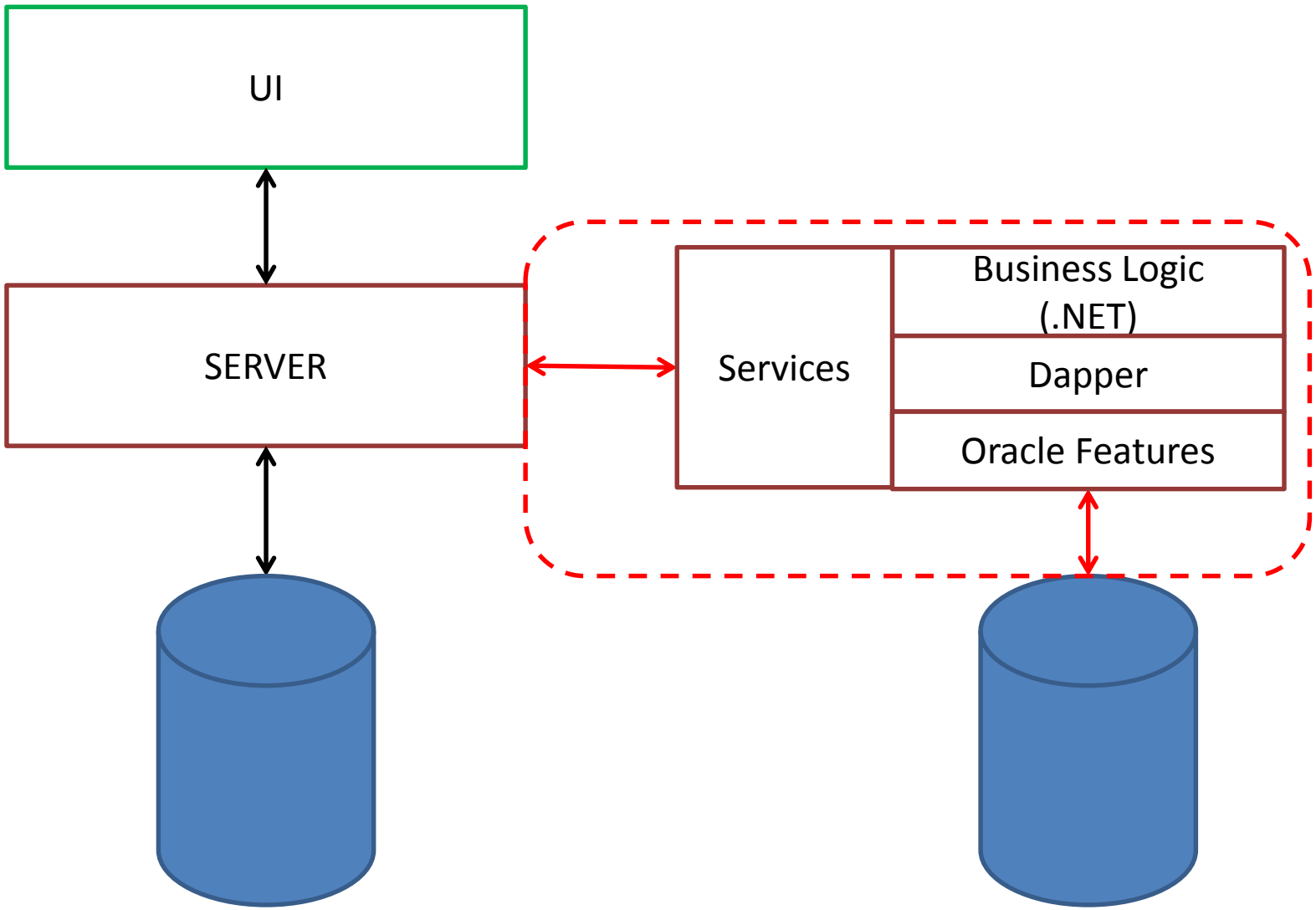
Как достичь результата?

- Избавиться от замедляющих факторов при работе с данными
- Максимальная экономия оперативной памяти
- Оптимизация процесса расчета с использованием возможностей C# в .NET
- Простое и наглядное использование кода

Стек технологий

- Dapper
- Логика расчета на .NET
- Oracle DB

РЕШЕНИЕ


UI
SERVER
Services
Business Logic
(.NET)
Dapper
Oracle Features

Dapper

Немного о Dapper

- Создана 7 лет назад Сэмом Саффроном
- «Микро ORM»
- Тонкая обертка вокруг ADO.NET
- По сути маппер, предназначенный для быстрой зачитки данных

Почему именно Dapper?

- Плюсы
  - Позволяет быстро обрабатывать сколь угодно сложные запросы
  - Избегает проблемы большинства ORM (лишние JOIN, внезапные LEFT JOIN)
  - Позволяет работать с «плоскими» сущностями
- Минусы
  - Это не совсем ORM, кроме маппинга, ничего в нем нет

Простой пример с dynamic

```
public IList<Order> GetOrders()
{
    var sql = "SELECT * FROM Order";
    using (var connection = new OracleConnection ("DataSource=SomeDBConnection"))
    {
        var orders = connection.Query(sql).Select(o =>
                    new Order
                    {
                        Id = o.Id,
                        Quantity = o.Quantity,
                        ClientId = o.ClientId
                    }).ToList();
    }
}
```

Простой пример с автомаппингом

```
public IList<Order> GetOrders()
{
    var sql = "SELECT * FROM Order";
    using (var connection = new OracleConnection
     ("DataSource=SomeDBConnection"))
    {
        var orders = connection.Query<Order>(sql).ToList();
    }
}
```

Проблема стандартной реализации

- Нужно постоянно оборачивать в
  using(…)
  {
  …
  }
- Дублируем код для каждого репозитория
- Для такого типа систем чаще всего требуется всего 1 соединение

Упрощаем жизнь базовым репозиторием

```
private IDbConnection CreateConnection(string connectionString = null) => new
    OracleConnection(connectionString ??
    _connectionStringsManager.GetConnectionString());

protected T InConnection<T>(Func<IDbConnection, T> runnable)
{
   using (var connection = CreateConnection())
   {
      connection.Open();
      return runnable(connection);
    }
}
```

Теперь можно делать так

```
public IList<Order> GetOrders() =>
    InConnection(connection =>
        connection. Query<Order>("SELECT * FROM Order"))
    .ToList();
```

А что с параметрами?

```
protected IEnumerable<T> Query<T>(string query, object parameters) =>
     InConnection(_ => _.Query<T>(query, parameters));

 protected IEnumerable<dynamic> Query(string query, object parameters) =>
     InConnection(_ => _.Query(query, parameters));


public IList<Order> GetOrdersByClientId(long id) =>
    InConnection(connection =>
        connection. Query<Order>(
       @"SELECT * FROM Order  WHERE ClientId = :id”,
       , new {id}))
    .ToList();
```

Аналогичный синтаксический сахар делаем для остальных методов Dapper

- Execute
- Query
- QueryFirst
- QueryFirstOrDefault
- QuerySingle
- QuerySingleOrDefault
- QueryMultiple

Все ли теперь хорошо?

Например, могут понадобиться такие конструкции:

```
public Dictionary<long, long[]> GetPairs() =>
    InConnection(connection =>
        connection. Query (
        @"SELECT Id, ClientId
        FROM Order o
        JOIN Client c
        ON c.Id = o.ClientId”))
        .Select(d => new
                                {
                                    ClientId = d.ClientId,
                                    OrderId = d.Id
                                 })
                        .ToArray().GroupBy(d => d.ClientId)
                        .ToDictionary(p => p.Key, p => p.ToArray());
```

Воспользуемся паттерном IdOf<T>

```
public struct IdOf<TEntity>: IEquatable<IdOf<TEntity>>    {
    public IdOf(long id) => Id = id;
    public long Id { get; }
    public static implicit operator long(IdOf<TEntity> id) => id.Id;
    public static bool operator ==(IdOf<TEntity> left, long right) {return left.Id == right;}
    public static bool operator !=(IdOf<TEntity> left, long right) {return left.Id != right;}
    public override string ToString() => Id.ToString();
    public override bool Equals(object obj) => !ReferenceEquals(null, obj) && obj is IdOf<TEntity> id &&
      Equals(id);
    public override int GetHashCode() =>  Id.GetHashCode();
    public bool Equals(IdOf<TEntity> other) => Id == other.Id;
}
```

Немного о IEquatable<T>

- Поможет избежать боксинга при сравнении структур
- Реализуется у всех примитивов-структур
- Поддерживается стандартными Generic коллекциями (вызывается через **IEqualityComparer<T>.Equals(T, T)**)
- Подробнее https://habr.com/post/315168/

Также будет полезно

```
public static class IdOfExtensions    {
    public static IdOf<TEntity> AsIdOf<TEntity>(this long id) => new IdOf<TEntity>(id);
     public static IdOf<TEntity> AsIdOf<TEntity>(this int id) => new IdOf<TEntity>(id);
}
```

Теперь можно делать так

```
public Dictionary<IdOf<Client>, IdOf<Order>[]> GetPairs() =>
    InConnection(connection =>
        connection. Query (
        @"SELECT Id, ClientId
        FROM Order o
        JOIN Client c
        ON c.Id = o.ClientId”))
        .Select(d => new
                                {
                                    Client = d.ClientId.AsIdOf<Client>(),
                                    Order = d.Id.AsIdOf<Order>()
                                 })
                        .ToArray().GroupBy(d => d.Client)
                        .ToDictionary(p => p.Key, p => p.ToArray());
```

Иные преимущества Dapper

- Можно подключать хэндлеры типов, что позволяет, например, маппить IdOf<T> как параметры
- Поддерживает асинхронные, буферизированные запросы
- Позволяет использовать транзакционность InTransaction() => …

Ускорение логики на C#

А что же делать с логикой на C#?

- Сколько раз сработает сборщик мусора при последовательной вычитке десятков миллионов строк?
- Сколько будут «весить» все полученные сущности?

Корень проблемы - объект сущности

- Каждый объект сущности будет жить в куче
- Относительно долгий процесс создания и инициализации
- Имеет оверхед (за счет ссылок на блок синхронизации и VMT)
  - 8 бит для 32х битной архитектуры
  - 16 бит для 64х битной архитектуры
- Adam Sitnik - Value Types vs Reference Types

Структуры как сущности

- Сами по себе не задействуют кучу, сокращая количество сборок мусора
- Обеспечивают экономию памяти машины в процессе расчета
- В огромном ряде случаев быстрее инициализируются данными (например, при вычитке и маппинге)
- Удобно копировать данные

Минусы такого подхода

- Нельзя переопределить конструктор по умолчанию
- Предпочтительней использовать плоскую реализацию для каждой отдельной структуры
- Нужно обеспечивать самостоятельный контроль боксинга
- Пробрасывать в большом стеке вызовов методов придется по ссылке, либо избегать таких конструкций в коде
- Нельзя выстраивать иерархии наследования

Стоит ли использовать структуры?

- В общем случае – нет, так как нарушается множество принципов для работы с сущностями в объектно ориентированном стиле
- Чревато массой неудобств
- Не все ORM хорошо заточены для работы с ними
- Структуры как сущности полезны только там, где это необходимо – для узкого круга задач
- .NET Value Type (struct) as a DDD Value Object

Оптимальное использование

- Зачитываем сущности в структуры
- Структуры создаем плоскими
- Все ID оборачиваем в паттерн IdOf<T>
- Не забываем реализовать IEquatable<T>
- Следим за боксингами, избегаем неправильных пробросов в параметрах.

Как еще можно ускорить расчет и обработку данных?

- Использовать параллельную обработку данных по пачкам, например, используя Parallel LINQ
- Иногда дает профит реализация приложения, где различные части расчета осуществляются последовательным запуском приложения с разными аргументами командной строки

Фишки Oracle

А что же с Oracle?

- OracleBulkCopy
- Merge

OracleBulkCopy

- Содержится в Oracle.DataAccess
- Позволяет крайне быстро массово копировать данные
- Игнорирует индексы, требует их восстановления
- Плохо дружит с триггерами и констрейнтами
- Требует особого подхода при использовании материализованных View

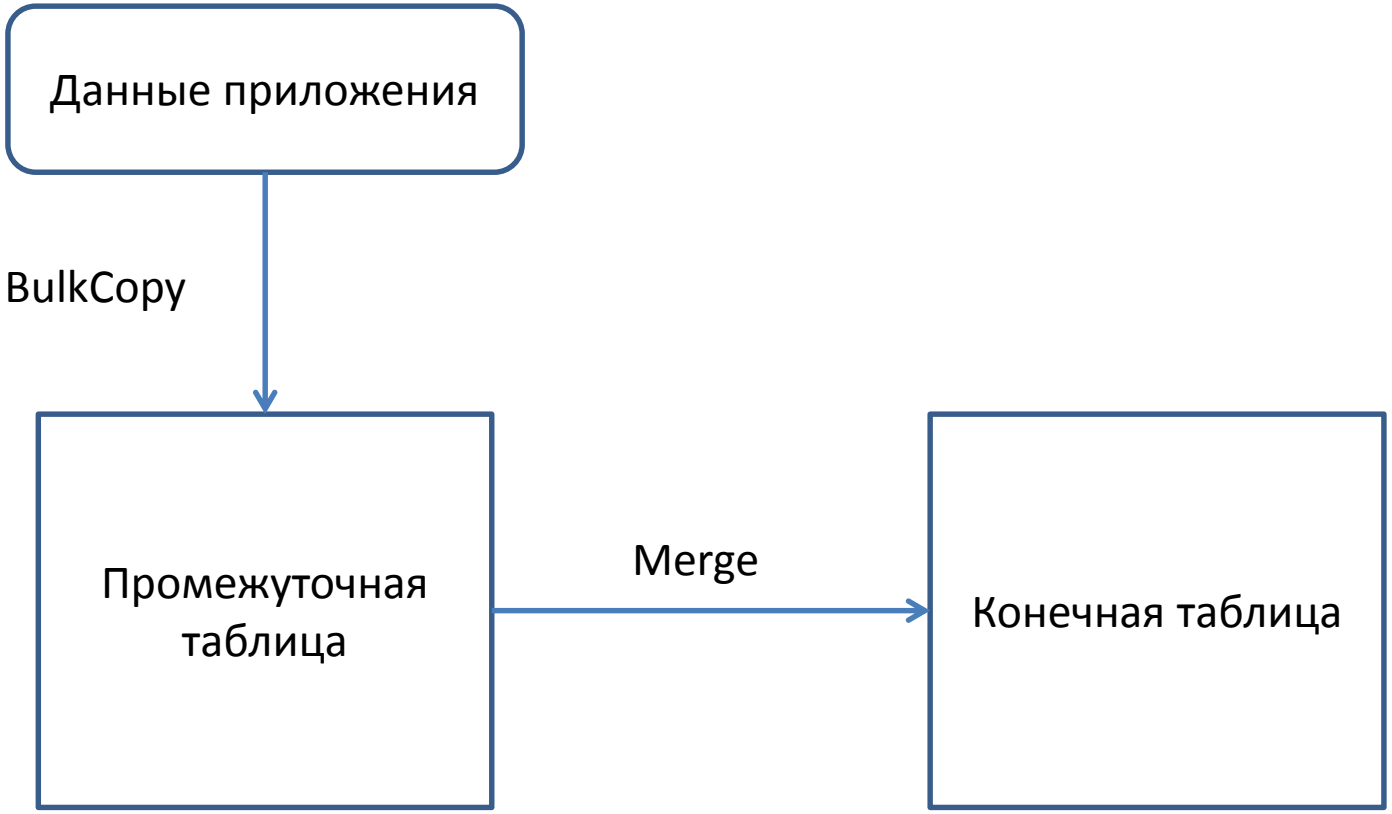
Алгоритм использования

- Создается DataTable, содержащий данные для копирования
- Truncate Table
- Сделать триггеры неактивными при необходимости
- Вызвать BulkCopy (можно поиграться с настройками)
- Восстановить индексы
- Активировать триггеры при необходимости

На что стоит обратить внимание

- Формирование ключей придется делать вручную, так как триггеры будут недоступны
- Данные в DataTable лучше всего располагать в том же порядке, что и в таблице БД
- Маппить Enum придется вручную
- Удобнее всего реализовывать в виде адаптера к базовому репозиторию
- Есть реализация BulkCopy для SqlServer (SqlBulkCopy)

Merge


Данные приложения
BulkCopy
Промежуточная таблица
Merge
Конечная таблица

Подведем итоги

Где это может быть полезно?

- Нужно быстро обработать и записать большое количество данных
- Есть необходимость обновлять данные с определенной периодичностью
- Данные используются на протяжение длительного периода сторонними приложениями
- Есть гарантия, что на период расчета пользователю данные будут не нужны

Резюме

- Dapper обеспечивает быстрый и удобный доступ к данным
  Это не совсем ORM, но и не голый Adapter
- При создании приложения с Dapper лучше всего использовать облегчающие жизнь конструкции
- Использование структур позволяет достичь больших плюсов в производительности и расходах памяти, если их использовать с умом, не допуская лишних боксингов
- Для быстрой записи данных используйте BulkCopy и Merge вместо update и insert

Благодарности коллегам

- Серавкину Всеволоду
- Щекочихиной Марии

Спасибо!

Вопросы?

Орлов Юрий

justice1786@gmail.com