Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Институт естественных и точных наук
Факультет математики, механики и компьютерных технологий
Кафедра прикладной математики и программирования
Направление подготовки Прикладная математика и информатика

РАБОТА ПРОВЕРЕНА

_____
«____»_____2018г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, д.ф.-м.н., доцент
_____/А.А.Замышляева
«____»_____2018 г.

Разработка библиотеки классов для преобразования, интегрированного в язык программирования, запроса SQL

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУрГУ–09.03.04.2018.108.ПЗ ВКР

Руководитель работы, доцент
_____/А.К. Демидов
«____»_____2018 г.
Автор работы
Студент группы ЕТ-414
_____/ Д.К. Ширман
«____»_____2018 г.
Нормоконтролер, доцент
_____/Т.Ю.Оленчикова
«____»_____2018 г.

Челябинск 2018

АННОТАЦИЯ

Ширман Д. К. Разработка библиотеки классов для преобразования, интегрированного в язык программирования, запроса SQL. – Челябинск: ЮУрГУ, ЕТ-414, 96 с., 9 ил., библиогр. список – 15 наим., 2 прил.

Данная работа посвящена разработке библиотеки классов для преобразования запросов SQL на языке C#.

В работе выполнен обзор существующих альтернативных решений, анализ требований к системе, а также определен минимальный необходимый функционал. Спроектирована и разработана архитектура системы, включающая в себя диаграмму классов. Проведена проверка работоспособности программы с помощью написания Unit-тестов и реализации demo проекта.

В приложениях приведены руководство пользователя и исходный код библиотеки.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

«Active Database Software» – является частной компанией по разработке программного обеспечения (ПО) для профессиональных пользователей баз данных (БД), администраторов и разработчиков.

Главная цель компании – облегчить поиск данных для пользователей баз данных, предоставляя средства для работы с SQL-запросами. Для этого были созданы графические инструменты, которые позволяют легко понять схему базы данных и визуально создавать запросы, а также создавать программные API-интерфейсы, которые позволяют анализировать и изменять SQL-запросы.

На текущий момент пользователи существующего API-интерфейса для программного создания обычного SQL-запроса, вынуждены писать большое количество кода, который не является корректным по структуре и диалекту относительно языка SQL. Поэтому возникает задача создать такой API, который позволит не только сократить время написания кода, но и будет соответствовать стандартам языка SQL, что в свою очередь позволит строить корректные запросы.

Целью данной работы является разработка библиотеки классов для преобразования запросов SQL на языке C#, которая в полной мере отражает основные функциональные возможности языка SQL.

Практическая значимость создаваемой библиотеки заключается в повышении эффективности разработки приложений, в основе которых лежит работа с БД, а также увеличение заинтересованности клиентов в использовании этого инструмента в своих проектах.

1 АНАЛИЗ ТРЕБОВАНИЙ. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ.

1.1 Постановка задачи

Поставлена задача разработать библиотеку классов для преобразования интегрированного в язык программирования запроса SQL. Библиотека предназначена для «типобезопасного» построения запроса к серверу баз данных.

Главной особенностью разрабатываемой библиотеки является:

- поддержка разных диалектов языка SQL;
- поддержка разных СУБД;
- генерация C# кода из структуры метаданных БД.

Целью создания ПО являются:

- повышение эффективности разработки приложений, в основе которых лежит работа с БД;
- повышение скорости создания запроса SQL;
- обеспечение целостности данных и предотвращение ввода некорректной информации.

Для достижения поставленной цели необходимо решить следующие задачи:

- выполнить анализ требований к программному обеспечению;
- провести обзор существующих решений;
- спроектировать архитектуру системы;
- описать основные алгоритмы;
- разработать ряд тестов для отладки и тестирования системы.

С учетом развития разных диалектов языка SQL и самого языка, необходимо добавлять новые функциональные возможности.

В дальнейшем планируется добавить возможность генерации метаданных подпрограмм (функции и процедуры). А также возможность представлять данные результата SQL-запроса в объектном виде.

1.2 Обзор существующих решений

Многие решения имеют аналогичные API-интерфейсы с похожими наборами функций.

1.2.1 JOOQ

JOOQ (Java Object Oriented Querying) – библиотека с открытым исходным кодом, которая предоставляет возможность сопоставления схемы БД с Java-кодом, а так же предметно-ориентированный язык (Domain specific language) для построения запросов используя сгенерированные классы.

Особенностью данной библиотеки является то, что с помощью JOOQ вы не только строите запрос, но и получаете объектные данные этого запроса прямо в коде.

Обзор application programming interface(API) модели.

JOOQ позволяет строить запросы в точно такой же манере как и обычные SQL-запросы к СУБД (см. рисунок 1.1).

SQL:

```
SELECT AUTHOR.FIRST_NAME,
AUTHOR.LAST_NAME, COUNT(*)
    FROM AUTHOR
    JOIN BOOK ON AUTHOR.ID =
BOOK.AUTHOR_ID
   WHERE BOOK.LANGUAGE = 'DE'
     AND BOOK.PUBLISHED > '2008-
01-01'
GROUP BY AUTHOR.FIRST_NAME,
AUTHOR.LAST_NAME
  HAVING COUNT(*) > 5
ORDER BY AUTHOR.LAST_NAME ASC
NULLS FIRST
   LIMIT 2
  OFFSET 1
     FOR UPDATE
```

Code:

```
DSLContext create = DSL.using(connection, dialect);

create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count())
      .from(AUTHOR)
      .join(BOOK).on(BOOK.AUTHOR_ID.eq(AUTHOR.ID))
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt("2008-01-01"))
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(2)
      .offset(1)
      .forUpdate()
      .fetch();
```

Рисунок 1.1 – Пример JOOQ API

На официальном сайте предоставлена документация, которая в полной мере описывает все возможности данного API интерфейса. Можно заметить, что для каждого SQL Clause разработчики выделили соответствующий методы:

1. *With Clause*

   Необязательная часть для оператора SELECT, где указываются общие табличные выражения (Common Tables Expressions).
2. *Select Clause*

   Предложение SELECT позволяет создавать собственные типы записей, ссылаясь на поля таблицы, функции, арифметические выражения и т.д.
3. *From Clause*

   Предложение SQL FROM позволяет указать любое количество табличных выражений для выбора данных.
4. *Join Clause*

   Предложение JOIN используется для объединения строк из двух или более таблиц на основе соответствующего столбца между ними.
5. *Where Clause*

Предложение WHERE используется для извлечения только тех записей, которые соответствуют заданному условию.

6. *Group By Clause*

   Оператор GROUP BY часто используется с агрегатными функциями (COUNT, MAX, MIN, SUM, AVG) для группировки результирующего набора одним или несколькими столбцами.

7. *Having Clause*

   Предложение HAVING было добавлено в SQL, потому что ключевое слово WHERE не могло использоваться с агрегатными функциями.

8. *Order By Clause*

   Ключевое слово ORDER BY используется для сортировки результирующего набора в порядке возрастания или убывания.

9. *Limit-Offset Clause*

   Предложение используется для указания количества возвращаемых записей.

10. *Union, Intersection and Except*

    Оператор UNION используется для объединения результирующего набора из двух или более операторов SELECT.

    Оператор EXCEPT возвращает уникальные строки из левого входного запроса, которые не выводятся правым входным запросом.

    Оператор INTERSECT возвращает уникальные строки, выводимые левым и правым входными запросами.

Особенностью JOOQ является унифицированный диалект, реализованный при помощи иерархии интерфейсов Java SQL, подходящий для многих диалектов. Эта концепция чрезвычайно эффективна при использовании JOOQ в современных IDE с завершением синтаксиса. Так же нужно отметить возможность генерации java классов из метаданных БД и использование их в качестве идентификаторов при построении запросов.

1.2.2 SqlKata

SqlKata – построитель запросов написанный на языке C#. Предоставляет возможность построения запроса посредством Fluent API, синтаксис которого очень похож на язык SQL. Он обеспечивает уровень абстракции над различными СУБД, что позволяет работать с несколькими базами данных с одним и тем же унифицированным API. SqlKata поддерживает сложные запросы, такие как вложенные условия, выбор из подзапроса, фильтрация по подзапросам, условные выражения, связи и т. п.

```
bool havingCommentsOnly = Config.Get('OnlyWithComments');

IEnumerable<Post> posts = await db.Query("Posts")
    .Where("Likes", ">", 10)
    .WhereIn("Lang", new [] {"en", "fr"})
    .WhereNotNull("AuthorId")
    .When(havingCommentsOnly, q => {

        // this will be executed only if `havingCommentsOnly` is true
        q.WhereExists(q => q.From("Comments").WhereColumns("PostId", "Posts.Id"))
    })
    .OrderByDesc("Date")
    .Select("Id", "Title")
    .GetAsync<Post>();
```

Рисунок 1.2 – Пример SqlKata API

Имеет аналогичную с JOOQ структуру запроса, подразделенную на предложения (Clauses), но как видно из примера (см. рисунок 1.2) нет поддержки иерархии интерфейсов, о которой упоминалось в ч. 1.2.1, что позволяет писать одни и те же методы несколько раз, делая API не схожим с SQL стандартом. Так же можно заметить, что все идентификаторы являются строковыми литералами из-за чего вероятность ошибки пользователя данного API остается прежней.

1.2.3 QueryDSL

QueryDSL Predicate — это мощный и чрезвычайно гибкий инструмент для работы с БД и просто подарок для Java-разработчиков, которые не очень хорошо разбираются в SQL (или совсем не разбираются), поскольку предикаты позволяют работать с БД при помощи привычного объектного представления сущностных зависимостей. Предикаты позволяют работать с элементами базы данных как с обычными полями класса. При сборке gradle создаёт специальные классы зависимостей, через которые и происходит поиск нужных записей в БД.

В QueryDSL есть 4 основных метода, позволяющих выдавать определённый результат:

- findOne() – позволяет искать какой-то один элемент. Вы должны быть уверены, что искомый элемент в БД только один, иначе дратути исключение;
- findAll() и несколько его перегрузок — возвращает iterable список найденных записей, отвечающих условиям. Обычно этот список потом приходится оборачивать в List (у нас это будет);
- count() – возвращает количество найденных элементов;

– exists() – возвращает булевское значение, существует такой элемент в таблице или нет.

Опираясь на документацию, можно выделить некоторые схожие черты с JOOQ:

– проект реализован на Java;
– типобезопасное построение запроса является основным принципом Querydsl. Запросы создаются на основе сгенерированных типов, которые отражают свойства ваших типов доменов. Также вызовы функции/метода строятся полностью безопасным образом, см. рисунок 1.3;
– согласованность - еще один важный принцип. Пути построения запросов и операции одинаковы во всех реализациях, а также интерфейсы запросов имеют общий базовый интерфейс.

```
List<Person> persons = queryFactory.selectFrom(person)
  .where(person.children.size().eq(
    JPAExpressions.select(parent.children.size().max())
                  .from(parent)))
  .fetch();
```

Рисунок 1.3 - Пример QueryDsl API

1.3 Анализ требований к системе

1.3.1 Требования к функциональным характеристикам

Разрабатываемое программное обеспечение должно обладать следующими функциями:

– генерация C# кода из метаданных БД (таблицы, поля, представления, подпрограммы);
– поддержка разных диалектов языка SQL;
– возможность подключения к разным СУБД;
– возможность построения запроса подобно языку SQL.

1.3.2 Требования к надежности

Отказы вследствие некорректных действий пользователя при взаимодействии с API недопустимы. Должна обеспечиваться обработка ошибок, которые могут быть вызваны вводом неверных значений или неправильным форматом данных. Все аварийные ситуации (из-за недопустимых или непредусмотренных действий пользователей) должны обрабатываться на программном уровне [1].

1.3.3 Условия эксплуатации

1.3.3.1 Требования к квалификации пользователя

Пользователь, работающий с ПО, должен являться администратором базы данных, иметь доступ ко всем базам и таблицам.

Предполагается, что пользователь обладает базовыми знаниями в области администрирования баз данных, знаком с языками SQL и C#.

1.3.3.2 Требования к составу и параметрам технических средств

Для подключения библиотеки классов, необходимы:

- среда разработки проектов на языке C#;
- установленные библиотеки .Net Framework версии не ниже 4.0;
- данные для подключения к СУБД или XML файл с описанием метаданных БД.

1.4 Выводы по разделу

В данном разделе была сформулирована цель данного проекта, поставлены соответствующие задачи для ее реализации, а также выделены некоторые особенности, которыми должно обладать разрабатываемое ПО. Проведен анализ существующих решений, выделены общие черты в программных интерфейсах, которые позволят правильно построить архитектуру ПО. В пункте 1.3 были определены основные требования к системе.

2 РАЗРАБОТКА БИБЛИОТЕКИ

2.1 Модель иерархии программных интерфейсов

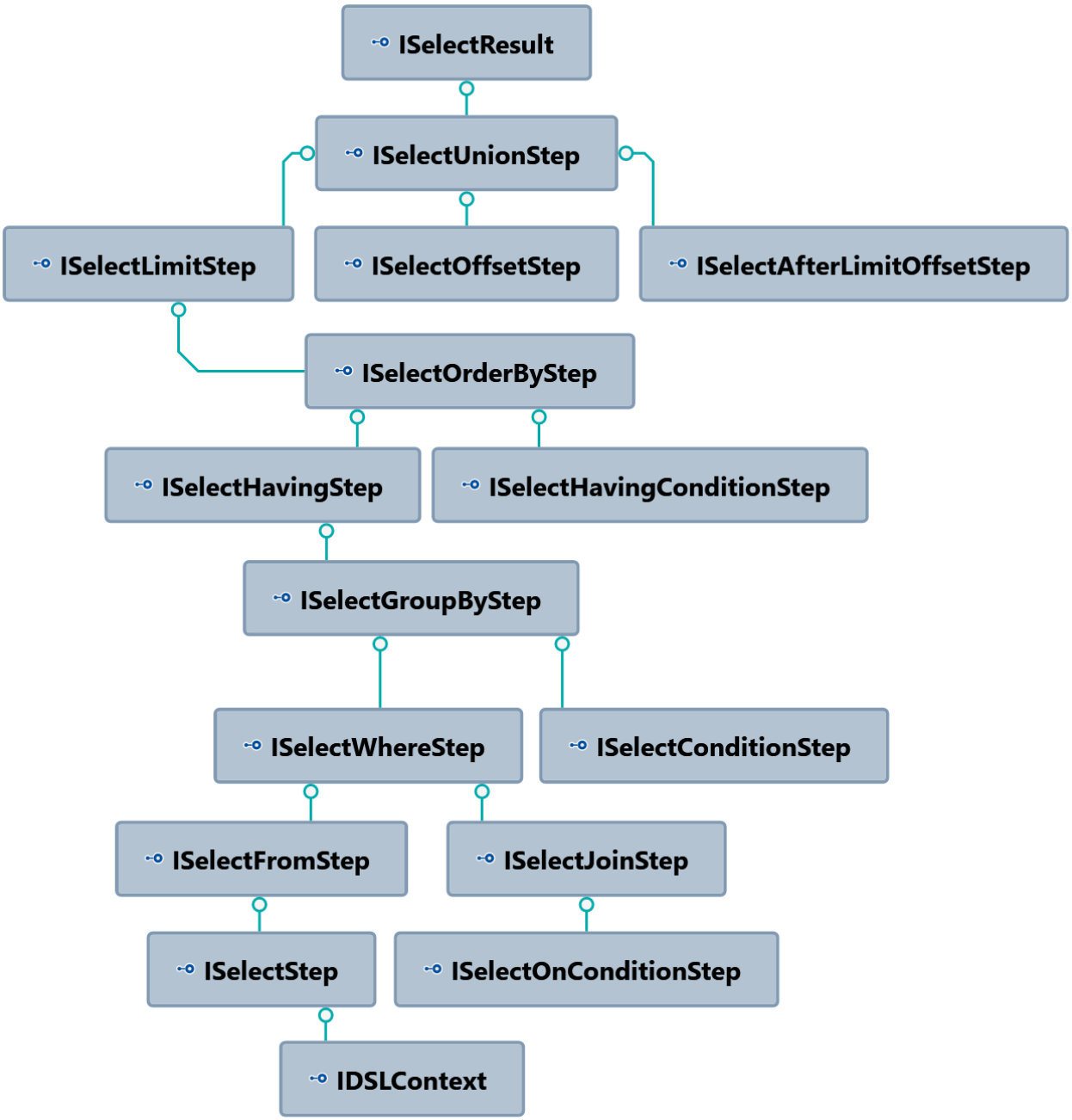

Рисунок 2.1 – Граф зависимостей интерфейсов

Опираясь на проведенный анализ (п. 1.2) было принято решение для каждого SQL Clause (Select, From, Where и т.д.) выделить интерфейс [2] с соответствующими методами и построить с помощью этих интерфейсов иерархию показанную на см. рисунок 2.1:

Можно заметить, что у большинства интерфейсов в названии присутствует постфикс Step (Шаг) [3], это подразумевает, что каждому вызову функции в стиле Fluent API соответствует свой шаг.

Пример:

```
public static string From_ex2()
{
   using (var create = DSL.Query)
      {
                var orders = Table("Orders");
             var result = create
                    .Select(orders.Field("OrderId")) //← SelectStep
                    .From(orders) //← FromStep
                    .Result;
         return Program.Formatted(result);
      }
}
```

2.2 Описание функционала интерфейсов

Интерфейс *ISelectResult*

```
public interface ISelectResult : IQueryStatisticsProvider, ITableLike, IFieldLike, ISelectItem
{
   QueryRoot Result { get; }
   string SQL { get; }
   ISelectResult As(string alias);
}
```

Общий интерфейс, который наследуют все интерфейсы [4] с постфиксом Step. Свойство Result с возвращаемым типом QueryRoot хранит в себе объектную модель запроса в виде дерева, см. рисунок 2.2:

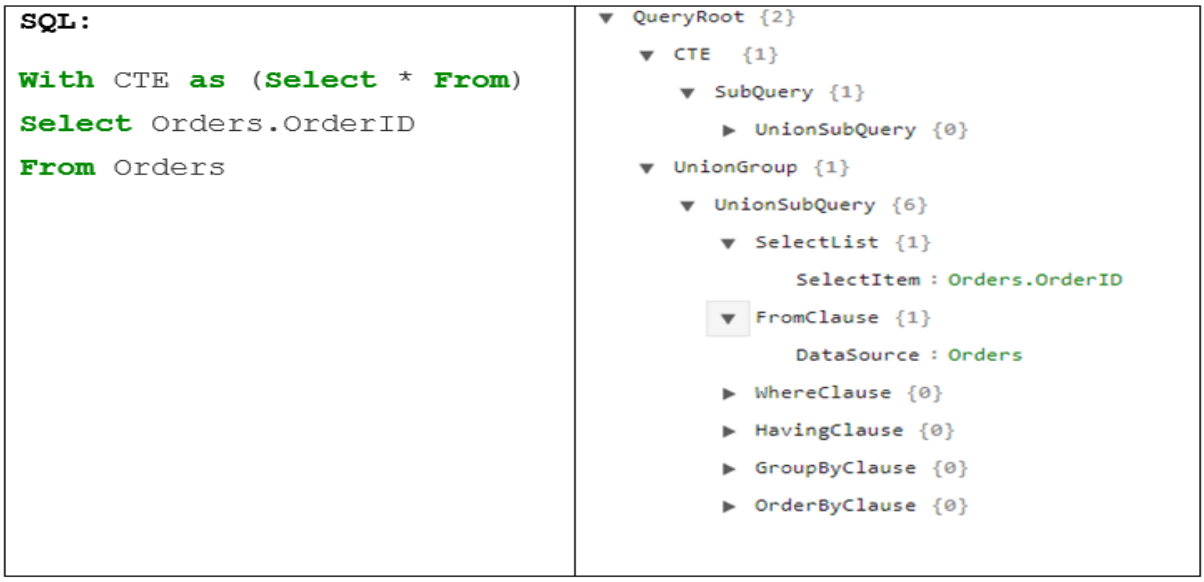

Рисунок 2.2 – Объектная модель запроса

Объектная модель

Чтобы получить объектную модель запроса пользователь должен присвоить свойству SQL строковое выражение, где и будет содержатся сам запрос SQL. Далее данное строковое выражние будет проанализировано и разбито на отдельные узлы абстрактного синтаксического дерева с помощью существующих синтаксических и лексических анализаторов, написанных с использованием программы COCO/R. Свойство SQL содержится в корне объектной модели (класс QueryRoot) оно доступно для чтения и для записи, а в интерфейсе ISelectResult только для чтения так как пользователь нашего API должен строить запрос частями сначала в предложении Select, затем From и т.д.

Coco/R — программа генерации компиляторов или интерпретаторов языка. Программа читает файл с атрибутивной грамматикой реализуемого языка, описанной в форме РБНФ (Расширенная форма Бэкуса — Наура, EBNF) и генерирует ряд файлов:

- исходники лексического анализатора (сканера); работает как конечный автомат.
- исходники синтаксического анализатора (парсера); использует методику нисходящего рекурсивного спуска.
- информационные файлы (лог, таблица лексем языка).

Использование Coco/R является очень простым. Создаваемый программой код является быстрым и лёгким для понимания. Разработка своего языка заключается в разработке файла грамматики языка. В грамматику языка добавляется специального вида комментарии (. .), в которых заключен код для выполнения дополнительных действий. Как правило, это код для занесения данных в таблицы идентификаторов, генерации кода или его интерпретации.

Получившееся дерево хранится в написанных уже вручную классах, которые и представляют для пользователя объектную модель запроса [5]:

- QueryRoot – корень модели;
- UnionGroup – группа соодержащая в себе подзапросы объединенные оператором UNION;
- UnionSubQuery – подзапрос содержащий в себе списки: Select, From, Where, Having, Group By, Order By.

Реализуя данный интерфейс, мы предоставляем возможность пользователю получить построенный с помощью API SQL-запрос на любом шаге либо в виде объектной модели, либо в виде строки (свойство SQL).

Интерфейс *ISelectUnionStep*:

```
public interface ISelectUnionStep : ISelectResult
{
ISelectOrderByStep Union(Action<ISelectStep> selectAction);
ISelectOrderByStep UnionAll(Action<ISelectStep> selectAction);
ISelectOrderByStep Union(ISelectResult select);
ISelectOrderByStep UnionAll(ISelectResult select);
}
```

Интерфейс, предоставляющий методы для объединения двух подзапросов подобно Union оператору в SQL-запросе. В данном интерфейсе есть 2 перегрузки, одна из которых принимает в качестве параметра ссылку на функцию, которая в результате вызова вернет нам объект, содержащий в себе SQL, другая перегрузка принимает сразу этот объект. Первый вариант удобен тем, что пользователю не приходится создавать новый объект для формирования запроса SQL, он создается внутри метода.

На этапе реализации данных методов стоит учесть ограничения, ктоторые описаны в стандарте SQL. Существуют два основных правила, регламентирующие порядок использования оператора Union:

- число и порядок извлекаемых столбцов должны совпадать во всех объединяемых запросах;
- типы данных в соответствующих столбцах должны быть совместимы.

Определения столбцов, данные из которых извлекаются в объединяемых запросах, не должны совпадать, однако должны быть совместимыми путём неявного преобразования. Если типы данных различаются, то получившийся тип данных определяется на основе правил очередности типов данных (для конкретной СУБД). Если типы совпадают, но различаются в точности, масштабе или длине, результат определяется на основе правил, используемых для объединения выражений (для конкретной СУБД) [6]. Типы не определенные ANSI, такие как DATA и BINARY, обычно должны совпадать с другими столбцами такого же нестандартного типа.

Интерфейс *ISelectLimitStep*

```
public interface ISelectLimitStep : ISelectUnionStep
{
    ISelectOffsetStep Limit(int val);
    ISelectOffsetStep Limit(string limit);
}
```

Интерфейс предоставляет методы для ограничения количества строк, выводимых после выполнения запроса. Эквивалент таких SQL операторов как TOP, LIMIT, ROWNUM (в зависимости от используемого диалекта).

Интерфейс *ISelectOffsetStep*

```
public interface ISelectOffsetStep : ISelectUnionStep
{
    ISelectUnionStep Offset(int val);
    ISelectUnionStep Offset(string offset);
}
```

Метод Offset предоставляет возможность пропустить заданное количество строк перед выводом результата SQL запроса.

Интерфейс *ISelectOrderByStep*

```
public interface ISelectOrderByStep : ISelectLimitStep
{
    ISelectLimitStep OrderBy(params string[] sortingItems);
    ISelectLimitStep OrderBy(params IOrderByItem[] orderByItems);
}
```

Интерфейс, предоставляющий методы для сортировки результирующего множества. Эквивалент Order by clause в SQL.

Необязательный (опциональный) параметр операторов SELECT и UNION, который означает что операторы SELECT, UNION возвращают набор строк, отсортированных по значениям одного или более столбцов. Его можно применять как к числовым столбцам, так и к строковым. В последнем случае, сортировка будет происходить по алфавиту.

Использование предложения ORDER BY является единственным способом отсортировать результирующий набор строк. Без этого предложения СУБД может вернуть строки в любом порядке. Если упорядочение необходимо, ORDER BY должен присутствовать в SELECT, UNION.

Сортировка может производиться как по возрастанию, так и по убыванию значений.

- параметр ASC (по умолчанию) устанавливает порядок сортировки по возрастанию, от меньших значений к большим;
- параметр DESC устанавливает порядок сортировки по убыванию, от больших значений к меньшим;

Если столбец результатов запроса, используемый для сортировки, является вычисляемым, то у него нет имени, которое можно указать в предложении сортировки. В таком случае вместо имени столбца необходимо указать его

порядковый номер или повторить выражение в предложении ORDER BY. Использование номера столбца - более старый способ, который не рекомендуется к применению, как более подверженный ошибкам (например, при изменении порядка столбцов в предложении SELECT).

Интерфейс *ISelectHavingStep*

```
public interface ISelectHavingStep : ISelectOrderByStep
{
    ISelectHavingConditionStep Having(Condition condition);
    ISelectHavingConditionStep Having(string condition);
}
```

Интерфейс, предоставляющий методы для задания условий агрегатным функциям в запросе. Эквивалент Having clause в SQL.

Необязательный (опциональный) параметр оператора SELECT для указания условия на результат агрегатных функций (MAX, SUM, AVG, …).

HAVING <условия> аналогичен WHERE <условия> за исключением того, что строки отбираются не по значениям столбцов, а строятся из значений столбцов, указанных в GROUP BY, и значений агрегатных функций, вычисленных для каждой группы, образованной GROUP BY [7].

Необходимо, чтобы в SELECT были заданы только требуемые в выходном потоке столбцы, перечисленные в GROUP BY и/или агрегированные значения. Распространённая ошибка — указание в SELECT столбца, пропущенного в GROUP BY.

Если параметр GROUP BY в SELECT не задан, HAVING применяется к «группе» всех строк таблицы, полностью дублируя WHERE (допускается не во всех реализациях стандарта SQL).

Интерфейс *ISelectWhereStep*

```
public interface ISelectWhereStep : ISelectGroupByStep
{
    ISelectConditionStep Where(string condition);
    ISelectConditionStep Where(Condition condition);
    ISelectConditionStep WhereExists(Action<ISelectStep> selectAction);
    ISelectConditionStep WhereExists(ISelectResult select);
    ISelectConditionStep WhereNotExists(Action<ISelectStep> selectAction);
    ISelectConditionStep WhereNotExists(ISelectResult select);
}
```

Интерфейс, предоставляет методы эквивалентные оператору Where в SQL для определения условия поиска строк, возвращаемых запросом. В данном

интерфейсе есть 2 перегрузки методов WhereExists и WhereNotExists, одна из которых принимает в качестве параемтра ссылку на функцию, которая в резуьтате вызова вернет нам объект, содержащий в себе SQL, другая перегрузка принмает сразу этот объект. Первый вариант удобен тем, что пользователю не приходится создавать новый объект для формирования запроса SQL, он создается внутри метода.

Интерфейс *ISelectGroupByStep*

```
public interface ISelectGroupByStep : ISelectHavingStep
{
    ISelectHavingStep GroupBy(params string[] groupingItems);
    ISelectHavingStep GroupBy(params IGroupingItem[] groupingItems);
}
```

Интерфейс предоставляет методы эквивалентные оператору Group By в SQL используемого с агрегатными функциями (COUNT, MAX, MIN, SUM, AVG) для группировки результирующего набора одним или несколькими столбцами.

Необходимо, чтобы в SELECT были заданы только требуемые в выходном потоке столбцы, перечисленные в GROUP BY и/или агрегированные значения.

SQL предоставляет нам возможность разбивать любую таблицу на логические группы (категории) и вычислять агрегатные статистические функции накаждой из этих групп. Для этого и служит предложение GROUP BY.

Интерфейс *ISelectConditionStep*

```
public interface ISelectConditionStep : ISelectGroupByStep
{
    ISelectConditionStep Or(string condition);
    ISelectConditionStep Or(Condition condition);
    ISelectConditionStep OrNot(string condition);
    ISelectConditionStep OrNot(Condition condition);
    ISelectConditionStep OrExists(string expression);
    ISelectConditionStep OrExists(Action<ISelectStep> selectAction);
    ISelectConditionStep OrExists(ISelectResult select);
    ISelectConditionStep OrNotExists(string expression);
    ISelectConditionStep OrNotExists(Action<ISelectStep> selectAction);
    ISelectConditionStep OrNotExists(ISelectResult select);
    ISelectConditionStep And(string condition);
    ISelectConditionStep And(Condition condition);
    ISelectConditionStep AndNot(string condition);
    ISelectConditionStep AndNot(Condition condition);
    ISelectConditionStep AndExists(string expression);
    ISelectConditionStep AndExists(Action<ISelectStep> selectAction);
    ISelectConditionStep AndExists(ISelectResult select);
    ISelectConditionStep AndNotExists(Action<ISelectStep> selectAction);
    ISelectConditionStep AndNotExists(ISelectResult select);
}
```

Интерфейс предоставляет методы эквивалентные логическим операторам в языке SQL для логического объединения нескольких условий.

Условия отбора

В SQL используется множество условий отбора, позволяющих эффективно и естественно созадвать различные типы запросов.

Пять предикатов (условий), определенных стандартом ANSI/ISO:

- сравнение – значение одного выражения сравнивается со значением другого;
- проверка на принадлженость диапозону;
- проверка наличи во множестве;
- проверка на соответсвие шаблону – проверяется соответсутет ли стоковое значение, содержащееся в столбце, определенному шаблону;
- проверка на равенство значению NULL.

Интерфейс *ISelectJoinStep*

```
public interface ISelectJoinStep : ISelectWhereStep
{
    ISelectJoinOnStep Join(ITable table);
    ISelectJoinOnStep RightJoin(ITable table);
    ISelectJoinOnStep LeftJoin(ITable table);
    ISelectJoinOnStep InnerJoin(ITable table);
    ISelectJoinOnStep FullJoin(ITable table);

    ISelectJoinOnStep Join(string table);
    ISelectJoinOnStep RightJoin(string table);
    ISelectJoinOnStep LeftJoin(string table);
    ISelectJoinOnStep InnerJoin(string table);
    ISelectJoinOnStep FullJoin(string table);

    ISelectJoinOnStep Join(IName name);
    ISelectJoinOnStep RightJoin(IName name);
    ISelectJoinOnStep LeftJoin(IName name);
    ISelectJoinOnStep InnerJoin(IName name);
    ISelectJoinOnStep FullJoin(IName name);
}
```

Интерфейс предоставляет методы эквивалентные оператору JOIN в SQL используемого для объединения строк из двух или более таблиц на основе соответствующего столбца между ними.

JOIN – оператор языка SQL, который является реализацией операции соединения реляционной алгебры. Входит в предложение FROM операторов SELECT, UPDATE и DELETE.

Операция соединения, как и другие бинарные операции, предназначена для обеспечения выборки данных из двух таблиц и включения этих данных в один результирующий набор. Отличительными особенностями операции соединения являются следующие:

- в схему таблицы-результата входят столбцы обеих исходных таблиц (таблиц-операндов), то есть схема результата является «сцеплением» схем операндов;
- каждая строка таблицы-результата является «сцеплением» строки из одной таблицы-операнда со строкой второй таблицы-операнда.

Определение того, какие именно исходные строки войдут в результат и в каких сочетаниях, зависит от типа операции соединения и от явно заданного условия соединения. Условие соединения, то есть условие сопоставления строк исходных таблиц друг с другом, представляет собой логическое выражение (предикат).

При необходимости соединения не двух, а нескольких таблиц, операция соединения применяется несколько раз (последовательно).

Интерфейс *ISelectJoinOnStep*

```
public interface ISelectJoinOnStep
{
    ISelectOnConditionStep On(string condition);
    ISelectOnConditionStep On(Condition condition);
}
```

Интерфейс предоставляет методы для задания условия объединения таблиц, указанных в JOIN Clause.

Для создания произвольного объединения предусмотрены два способа:

- применить синтаксис JOIN;
- применить синтаксис WHERE.

Стандарт ANSI SQL-92 предписывает применять синтаксис JOIN, но более старые стандарты SQL рекомендуют применять WHERE. Отсюда следует, что оба синтаксиса законны для большинства коммерческих СУБД.

Интерфейс *ISelectFromStep*:

```
public interface ISelectFromStep : ISelectWhereStep
{
    ISelectJoinStep From(params string[] dataSources);
    ISelectJoinStep From(params IFromClauseItem[] dataSources);
}
```

Интерфейс предоставляет методы эквивалентные оператору FROM в SQL для указания источников данных.SQL From является источником набора строк, который будет использоваться в Data Manipulation Language (DML). Данное предложение предоставит набор строк, который будет отображаться с помощью оператора Select, источника значений в операторе Update и целевых строк, которые будут удалены в инструкции Delete.

Интерфейс *ISelectStep*

```
public interface ISelectStep : ISelectFromStep
{
    IQueryModifier QueryModifier { get; }
    UnionSubQuery  ActiveUnionSubQuery { get; }
    ISelectStep Select();
    ISelectStep Select(params string[] strSelectItems);
    ISelectStep Select(params ISelectItem[] selectItems);
    ISelectStep SelectDistinct(params ISelectItem[] selectItems);
}
```

Интерфейс предоставляет методы эквивалентные оператору SELECT в SQL для выбора данных из указанных источников.

Оператор возвращает ноль или более строк. Список возвращаемых столбцов задается в части оператора, называемой предложением SELECT. Поскольку SQL является декларативным языком, запрос SELECT определяет лишь требования к возвращаемому набору данных, но не является точной инструкцией по их вычислению. СУБД транслирует запрос SELECT в внутренний план исполнения («query plan»), который может различаться даже для синтаксически одинаковых запросов и от конкретной СУБД.

Интерфейс *IDSLContext*

```
public interface IDSLContext : ISelectStep
{
    IWithAsStep With(string alias);
    IDSLContext With(ICommonTableExpression cte);
}
```

Интерфейс, являющийся корнем дерева иерархии интерфейсов, предоставляет методы для задания общих табличных выражений эквивалентно WITH Clause в SQL. Данный интерфейс, по сути, описывает базовый функционал унифицированного диалекта, который подходит для всех SQL диалектов.

Интерфейсы для объектов БД

Интерфейс *INamespace*

К пространствам имен относятся: сервер, пакет, схема, база данных. Схема базы данных (от англ. Database schema) — её структура, описанная на формальном языке, поддерживаемом СУБД. В реляционных базах данных схема определяет таблицы, поля в каждой таблице (обычно с указанием их названия, типа, обязательности), и ограничения целостности (первичный, потенциальные и внешние ключи и другие ограничения).

```
public interface INamespace
{
    /// <summary>
    /// Возвращает название пространства имен.
    /// </summary>
    string Name { get; }
 /// <summary>
    /// True – если это пространство имен используется по умолчанию
    /// </summary>
    bool Default { get; set; }
}
```

Схемы в общем случае хранятся в словаре данных. Хотя схема определена на языке базы данных в виде текста, термин часто используется для обозначения графического представления структуры базы данных.

Основными объектами графического представления схемы являются таблицы и связи, определяемые внешними ключами.

Пакет — это объект базы данных, который группирует логически связанные типы, программные объекты и подпрограммы SQL. Пакеты обычно состоят из двух частей, спецификации и тела, хотя иногда в теле нет необходимости.

Спецификация пакета – это интерфейс с вашими приложениями; она объявляет типы, переменные, константы, исключения, курсоры и подпрограммы, доступные для использования в пакете. Тело пакета полностью определяет курсоры и подпрограммы, тем самым реализуя спецификацию пакета.

Интерфейс *ITable*

Данный интерфейс реализуют сгенерированные классы таблиц и представлений.

Таблица – специальный тип данных SQL, который может быть использован для хранения результирующего набора для обработки в будущем. Тип table используется в первую очередь для временного хранения набора строк, возвращаемых как результирующий набор функции с табличным значением.

В общем ел учае в инструкциях SQL везде, где должно использоваться имя таблицы, можно использовать ее квалифицированное имя. Стандарт ANSl/ ISO SQL еще больше обобщает понятие квалифицированного имени таблицы. Он разрешает создавать именованное множество таблиц, называемое схемой. Вы можете обращаться к таблице определенной схемы с испюльзованием квалифицированного имени.

Представление — это виртуальная таблица, содержимое которой определяется запросом. Как и таблица, представление состоит из ряда именованных столбцов и строк данных. Пока представление не будет проиндексировано, оно не существует в базе данных как хранимая совокупность значений. Строки и столбцы данных извлекаются из таблиц, указанных в определяющем представление запросе и динамически создаваемых при обращениях к представлению.

```
public interface ITable : ITableLike
{
    /// <summary>
    /// Имя таблицы
    /// </summary>
    string TableName { get; }
    /// <summary>
    /// Схема базы данных в которой находится данная таблица
    /// </summary>
    string Schema { get; }
    /// <summary>
    /// Псевдоним
    /// </summary>
    string Alias { get; set; }
    /// <summary>
    /// Выражение выбора всех полей таблицы
    /// </summary>
    /// <returns></returns>
    string SelectAllFields();
}
```

Интерфейс *Ifield*

Интерфейс IField используется для реализации полей таблиц, представлений, подзапросов, общих табличных выражений.

Если в SQL-инструкции указано имя столбца, обычно SQL сам в состоянии определить, в какой из указанных в этой же инструкции таблиц содержится данный столбец. Однако если в инструкцию требуется включить два столбца из различных таблиц, но с одинаковыми именами, необходимо указать квалифицированные имена столбцов, которые однозначно определяют их местонахождение. Такое квалифицированное имя столбца состоит из имени таблицы, содержащей столбец, и имени столбца, разделенных точкой. Квалифицированное имя столбца можно использовать вместо короткого имени в инструкциях SQL там, где используется простое (неквалифицированное) имя.

Чтобы создать псевдоним произвольного столбца, можно воспользоваться методом AS. Любой псевдоним столбца – это какое-нибудь имя (альтернативное, вспомогательное, идентификатор), которое вы указываете для того, чтобы управлять тем, как заголовки столбцов отображаются при выдаче результатов.

```
        public interface IField : ISelectItem, IGroupingItem, IOrderByItem,
IConditionFactory, IAggregateFactory, ISortingFactory, IExpression<IField>
        {
            /// <summary>
            /// Имя поля
            /// </summary>
            string FieldName { get; }
            /// <summary>
            /// Имя поля с псевдонимом таблицы
            /// </summary>
            string NameInQuery { get; }
            /// <summary>
            /// Псевдоним таблицы
            /// </summary>
            string TableAlias { get; }
            /// <summary>
            /// Создать копию поля с новым <paramref name="alias"/>
            /// </summary>
            /// <param name="alias">Псевдоним</param>
            /// <returns>Новое поле типа <see cref="T"/></returns>
            new IField As(string alias);
        }
```

Интерфейс *ISubRoutine*

```
        public interface ISubRoutine
        {
            /// <summary>
            /// Возвращает имя подпрограммы
            /// </summary>
            string Name { get; }
            /// <summary>
            /// Возвращает пространство имен в котором находится данная
подпрограмма
            /// </summary>
            INamespace Schema { get; }
            /// <summary>
            /// Задает значение параметру подпрограммы
            /// </summary>
            void SetParameter(string name, object value);
            /// <summary>
            /// Список входных\выходных параметров подпрограммы
            /// </summary>
            List<IParameter> Parameters { get; }
        }
```

Данный интерфейс реализуют сгенерированные классы хранимых процедур и функций.

Храниимая процедура [8] — объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Хранимые процедуры очень похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. В хранимых процедурах могут выполняться стандартные операции с базами данных (как DDL, так и DML). Кроме того, в хранимых процедурах возможны циклы и ветвления, то есть в них могут использоваться инструкции управления процессом исполнения.

Хранимые процедуры похожи на определяемые пользователем функции (UDF). Основное различие заключается в том, что пользовательские функции можно использовать, как и любое другое выражение в SQL запросе, в то время как хранимые процедуры должны быть вызваны с помощью функции CALL:

Хранимые процедуры могут возвращать множества результатов, то есть результаты запроса SELECT. Такие множества результатов могут обрабатываться, используя курсоры, другими сохранёнными процедурами, возвращая указатель результирующего множества, либо же приложениями. Хранимые процедуры могут также содержать объявленные переменные для обработки данных и курсоров, которые позволяют организовать цикл по нескольким строкам в таблице. Стандарт SQL предоставляет для работы выражения IF, LOOP, REPEAT, CASE и многие другие. Хранимые процедуры могут принимать переменные, возвращать результаты или изменять переменные и возвращать их, в зависимости от того, где переменная объявлена.

Пользовательские функции сходны с хранимыми процедурами, но, в отличие от них, могут применяться в запросах так же, как и системные встроенные функции. Пользовательские функции, возвращающие таблицы, могут стать альтернативой просмотрам. Просмотры ограничены одним выражением SELECT, а пользовательские функции способны включать дополнительные выражения, что позволяет создавать более сложные и мощные конструкции.

2.3 Реализация интерфейсов

В пункте 1.2 мы определились с основными SQL Clauses и методами, которые соответствуют каждому шагу. Соответственно будет логично уделить особое внимание реализации основных методов таких как Select, From, Where, Having, GroupBy и OrderBy.

Метод Select

В качестве аргумента метод принимает один или несколько объектов, реализующих интерфейс ISelectItem. В данном интерфейсе содержится единственный метод AddToSelectList, который позволяет добавить элемент в запрос. Таким способом мы абстрагируемся от конкретных реализаций добавления элементов в список выбранных полей [9].

Перед тем как мы добавляем элемент в список мы сохраняем все текущие источники данных, которые уже содержатся в запросе и сохраняем источники данных после добавления поля, тем самым вычисляем те источники, которые добавились автоматически.

```
public ISelectStep Select(params ISelectItem[] selectItems)
{
   if (selectItems == null)
            throw new ArgumentNullException("selectItems");
      // Collect dataSources which already exists in query
      var previousDataSourceSet = new HashSet<DataSourceBase>();
      _activeUsq.FromClause.Items.ForEach(ds => previousDataSourceSet.Add(ds));

      foreach (var selectItem in selectItems)
      {
         selectItem.AddToSelectList(_queryModifier, _activeUsq);
      }
      // Collect dataSources after adding selectItems
      var currentDataSourceSet = new HashSet<DataSourceBase>();
      _activeUsq.FromClause.Items.ForEach(ds => currentDataSourceSet.Add(ds));
      // and check if it contains new dataSources
      // add them to _autoAddedDataSources
      _autoAddedDataSources
   .Append(currentDataSourceSet.Subtract(previousDataSourceSet));
   return this;
}
```

Это сделано потому, что текущая объектная модель запроса уже имеет в себе некоторую логику, которая автоматически добавляет таблицу в запрос если видит, что список выбранных полей содержит одно или несколько полей таблицы.

Метод From

В качестве аргумента метод принимает один или несколько объектов, реализующих интерфейс IFromClauseItem [10]. В данном интерфейсе содержится единственный метод AddToFromClause, который позволяет добавить источник данных в запрос. Данный интерфейс реализуют не только сгенерированные таблицы и представления из схемы БД, но и производные таблицы и подзапросы, которые тоже являются неявными источниками данных.

В данном методе все автоматически добавленные источники данных перемещаются на то место в запросе, которое указал пользователь при использовании API. Если пользователь использовал поле таблицы в запросе, но не добавил таблицу во From clause, таблица добавится автоматически. Так же можно заметить, что из метода уже возвращается новый экземпляр класса SelectJoinStep,

куда передается последний добавленный источник данных для того, чтобы можно было в дальнейшем создать связь между последним и новым источником.

```
    public ISelectJoinStep From(params IFromClauseItem[] items)
    {
       if (items == null)
                    throw new ArgumentNullException("items");
           DataSource last = null;
           for (int i = 0; i < items.Length; i++){
                     var currentDataSource =
items[i].AddToFromClause(_queryModifier, _activeUsq);
                  // If current dataSource was added automatically
                  if (_autoAddedDataSources.Contains(currentDataSource))
                  // then move it according to items position
                  {
                        _activeUsq.FromClause.Move(currentDataSource, i);
                        // delete this dataSource from _autoAddedDataSources
                        _autoAddedDataSources.Remove(currentDataSource);
                  }
                 // Save last dataSource
                 if (i == items.Length - 1)
                     last = currentDataSource;
             }
           Debug.Assert(last != null);
           return new SelectJoinStep(this, last);
    }
```

Метод Join

```
    public ISelectJoinOnStep Join(ITable table)
    {
       return CreateJoinFromTableHelper(table, JoinType.Join);
    }

    private ISelectJoinOnStep CreateJoinFromTableHelper(ITable table, JoinType
joinType)
    {
       var rightDataSource = table.AddToFromClause(_dslContext.QueryModifier,
_dslContext.ActiveUnionSubQuery);

          return new SelectJoinOnStep(_dslContext, _dataSource, rightDataSource,
joinType);
    }
```

В качестве аргумента метод принимает единственный объект реализующий интерфейс ITable. Данный интерфейс реализуют не только сгенерированные

таблицы схемы БД, но и представления, а также общие табличные выражения и производные таблицы. Главная задача этого метода совпадает с реализацией метода From, мы просто добавляем таблицу и передаем созданный источник в другой класс, в котором укажем условие связи двух объединенных таблиц. Данный метод существует в нескольких экземплярах, где в метод CreateJoinFromTableHelper передаются разные значения перечисления. Данное перечисление имеет 5 значений, указывающих тип связи. Соответственно остальные 4 экземпляра метода RightJoin, LeftJoin, InnerJoin и FullJoin имеют в своем имени значение перечисления, что сделано для удобства и сокращения написания кода. Метод Join, по сути, добавляет еще один источник данных в запрос и передает левый и правый источники в класс SelectJoinOnStep, где в дальнейшем пользователь укажет по какому критерию, будут объединены источники.

Метод Where

```
    public ISelectConditionStep Where(ConditionBase condition)
    {
       if (condition == null)
                    throw new ArgumentNullException("condition");
          var conditionsList = new ConditionsList(_sqlContext){condition};
          var conditionStep = new SelectConditionStep(this,
_activeUsq.WhereConditions,
conditionsList);
          _beforeGetResultConditionsActions
       .Add(conditionStep.AddConditionsToParent);
          return conditionStep;
    }
```

Метод принимает в качестве аргумента условие, по которому будут фильтроваться выбранные поля.

Особенность реализации данного метода заключается в том, что он не добавляет условия в объектную модель по факту вызова метода, а только когда пользователь захочет получить SQL в виде строки или самой модели. Такой подход обоснован тем, что список условий в объектной модели реализован в виде дерева (см. рисунок 5) и чтобы элементы были объединены корректно необходимо хранить временный список условий, с которым мы будем работать пока пользователь добавляет условия. И как только пользователь запросит SQL добавить временный список в запрос.

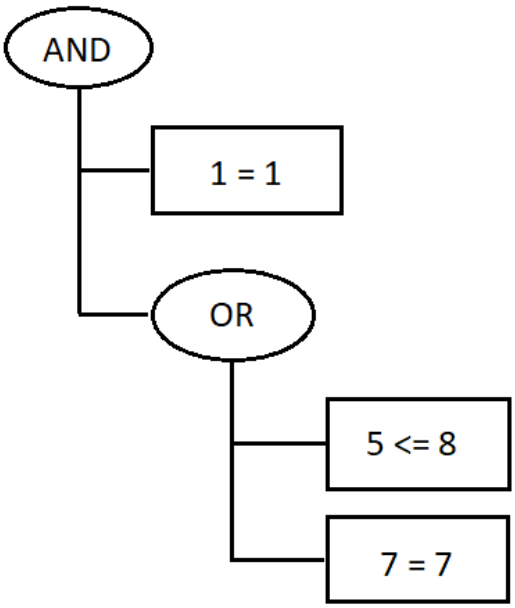

Рисунок 2.3 – Список условий

Метод Having

Реализован аналогично методу Where, где также список представлен в виде дерева и все условия добавляются в запрос по факту запроса SQL.

```
public ISelectHavingConditionStep Having(ConditionBase condition)
{
      if (condition == null)
         throw new ArgumentNullException("condition");

      var conditionsList = new ConditionsList(_sqlContext) {condition};
      var conditionStep =  new SelectHavingConditionStep(this,
ActiveUnionSubQuery.HavingConditions, conditionsList);

_beforeGetResultConditionsActions.Add(conditionStep.AddConditionsToParent);

      return conditionStep;
}
```

Метод GroupBy

Метод добавляет элементы, по которым в результате запроса будут сгруппированы выбранные поля. В качестве аргумента метод принимает один или несколько объектов, реализующих интерфейс IGoupingItem, который добавляет элемент в список элементов группировки запроса.

```
public ISelectHavingStep GroupBy(params IGroupingItem[] groupingItems)
{
   if (groupingItems == null)
                throw new ArgumentNullException("groupingItems");

      foreach (var groupingItem in groupingItems)
      {
         groupingItem.AddToGroupByList(_queryModifier, ActiveUnionSubQuery);
      }

      return this;
}
```

```
public ISelectLimitStep OrderBy(params IOrderByItem[] orderByItems)
{
   if (orderByItems == null)
                throw new ArgumentNullException("orderByItems");

      foreach (var orderByItem in orderByItems)
      {
         orderByItem.AddToOrderByList(QueryModifier, ActiveUnionSubQuery);
      }

      return this;
}
```

Метод OrderBy

Метод предназначен для добавления элементов, по которым в результате запроса будут отсортированы выбранные поля. В качестве аргумента принимает один или несколько объектов, реализующих интерфейс IOrderByItem, содержащий единственный метод AddToOrderByList.

Почти в каждом описанном выше методе все элементы списков реализовывали интерфейс с помощью, которого элемент сам мог добавить себя в список. Можно заметить, что сигнатуры у интерфейсных методов одинаковы. Первый параметр — это сервис который реализует логику низкоуровневого добавления элемента в объектную модель запроса, а второй параметр — это сама модель (текущий подзапрос).

Реализация интерфейсов для объектов БД

Проведя анализ было принято решение выделить для каждого объекта БД свой интерфейс, реализовать базовую функциональность в абстрактном классе, а затем при генерации схемы БД наследовать сгенерированные классы от абстрактных.

Пространства имен

Напомним, что в СУБД присутствуют такие понятия как: сервер, пакет, схема и база данных**.** Все эти понятия являются пространствами имен, которые хранят в себе элементы БД. Поэтому при генерации этих объектов сгенерированные классы будут реализовывать интерфейс INamespace и будут унаследованы от абстрактно класса AbstractNamespace.s

Таблицы и представления

Для таких объектов как таблицы и представления был написан интерфейс ITable (п. 2.2 Интерфейсы для объектов БД). Вся базовая функциональность была реализована в классе Table<T>. Общий параметр T здесь присутствует для того, чтобы при генерации класса конкретной таблицы передать в этот параметр название класса наследника (генерируемого класса) см. пример 1.

Пример 1:

```
[Name("Orders")]
class Orders : Table<Orders>
{
      private readonly TableField<int> _orderId;
      private readonly TableField<DateTime> _orderDate;
   ....
}
```

Таким образом при использовании метода As(alias) интерфейса ITable<T> мы получим объект класса Orders, а не его базовый класс Table.

Поля таблиц

Для полей таблиц и представлений был написан интерфейс IField<T> и создан базовый класс TableField<T>. В сгенерированном классе таблицы при определении поля, _orderDate (см. пример 1), вместо обобщённого параметра будет подставлен тот тип, который был определён в БД, в данном случае это тип DateTime. В .NET существует перечисление со всеми типами, которые могут присутствовать в СУБД и каждому значению мы поставили в соответствие один или более примитивных типов языка C#, пример 2.

Чаще всего поля таблиц используются в связке с агрегатными функциями, логическими и арифметическими операторами. Поэтому для удобства пользователя базовый класс поля таблицы наследует функциональность классов AggregatedFactory, SortingFactory и ConditionFactory. Данные классы представляют реализацию достаточно известного шаблона проектирования – абстрактная фабрика [11].

Пример 2:

```
public static Type DbTypeToType(DbType type)
{
      if (type == DbType.AnsiString) return typeof(System.String);
      else if (type == DbType.AnsiStringFixedLength) return
   typeof(System.String);
      else if (type == DbType.Binary) return typeof(System.Byte[]);
      else if (type == DbType.Boolean) return typeof(System.Boolean);
      else if (type == DbType.Byte) return typeof(System.Byte);
      else if (type == DbType.Currency) return typeof(System.Decimal);
   ....
}
```

Подпрограммы

Для генерируемых подпрограмм был написан интерфейс ISubRoutine (п. 2.2) он содержит в себе имя подпрограммы, ссылку на пространство имен в котором содержится данная подпрограмма и список входных\выходных параметров. Базовая функциональность реализована в абстрактном классе AbstractSubRoutine. Каждый класс после генерации содержит в себе всю необходимую информацию из БД. И при запуске приложения вся информация автоматически заполнит класс MetadataContainer, реализация которого содержится в библиотеке ActiveQueryBuilder.Core и была написана другими разработчиками. Его основная задача – хранить дерево метаданных схемы БД.

Для того, чтобы сгенерированные классы могли хранить имя конкретного объекта, БД был реализован класс NameAttribute, унаследованный от базового класса Attribute .Net framework. Что позволяет писать в коде проекта, см. пример 3:

Пример 3:

```
[Name("Customers")]
class Customers: Table<Customers>
[Name("dbo")]
class Dbo : AbstractNamespace
```

Далее в базовых классах объектов БД мы можем получить данное имя методом GetQualifiedName, которому в качестве аргумента передается текущий SQL контекст и с помощью рефлексии [12] мы извлекаем данный атрибут у текущего экземпляра класса, см. пример 4:

Пример 4:

```
    private string GetQualifiedName(SQLContext ctx)
    {
          using (var parsedName = new SQLQualifiedName(ctx))
          {
                parsedName.AddName(GetName(), true);
             return parsedName.QualifiedName;
      }
    }
    private string GetName()
    {
       var nameAttribute =
(NameAttribute)GetType().GetCustomAttributes(typeof(NameAttribute),
false).FirstOrDefault();
          return nameAttribute.Name;
    }
```

2.4 Реализация генерации C# кода

Для того чтобы сгенерировать код на языке C# были использованы классы пространства имен System.CodeDOM [13] библиотеки System.dll входящей в пакет .NET Framework.

Для того чтобы пользователь смог правильно сгенерировать структуру БД был спроектирован интерфейс класса конфигурации для генерации метаданных базы данных. Он устанавливает такие свойства как:

- Тип SQL диалекта
- Тип провайдера метаданных
- Тип подключения к БД
- Путь к файлу со схемой БД (если не установлен провайдер)
- Строка подключения к БД (если не указан путь к схеме)
- Путь к директории куда будут сгенерирован C# код.
- Название пространства имен в котором будут находиться сгенерированные классы.
- Специальный объект, который задает фильтр метаданных

GenerationTool класс который был реализован для текущей задачи. Он читает конфигурационный файл и файл со схемой БД (если был указан провайдер файл схемы БД будет сгенерирован «на лету»). Нужно отметить что файл схемы во время выполнения программы преобразуется в MetadataContainer, который хранит всю необходимую информацию о схеме БД.

Описание конфигурационного файла

Секция <configSections>

Регистрация секции конфигурации.

Определяет, каким классом (атрибут type) будет представлена конфигурация в коде проекта. Атрибут name = ’Имя_зарегестрированной_секции’, имя может быть любым.

```
    <configuration>
        <configSections>
            <section name="generation"
type="ActiveQueryBuilder.Core.MetadataCodeGeneration.GeneratorConfig,
ActiveQueryBuilder.Core"/>
        </configSections>
        <generation>
            <syntaxProvider type="ActiveQueryBuilder.Core.MSSQLSyntaxProvider,
ActiveQueryBuilder.Core" />
            <metadataProvider type="ActiveQueryBuilder.Core.MSSQLMetadataProvider,
ActiveQueryBuilder.MSSQLMetadataProvider"/>
            <metadataSource xml="D:\tmp\NorthwindWithFuncs.xml">
                <dbConnection type="System.Data.SqlClient.SqlConnection,
System.Data, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
connectionString="Data Source=MY-PC\SQLEXPRESS;Initial Catalog=db_main;Integrated
Security=True" />
            </metadataSource>
            <target namespace="Sakila"
directory="D:\tmp\TestProj\TestProj\SakilaDB"/>
        </generation>
```

Секция <’Имя_зарегестрированной_секции’>

Обязательный тег <*syntaxProvider*>, определяющий SQL диалект, в атрибуте type указываем полное имя сборки соответствующей библиотеки.

Тег <*metadataSource*>:

- атрибут *xml* - абсолютный путь к вашему файлу метаданных.

Тег <*dbConnection*> - конфигурация соединения с базой данных в реальном времени.

- атрибут *type* – полное имя класса для установки соединения;
- атрибут *connectionString* – строка подключения к вашей БД.

Замечания: если присутствует тег *dbConnection*, то атрибут *xml* нужно удалить.

Тег <*target*>:

- атрибут *namespace* – имя пространства имен, в котором будут находиться все сгенерированные классы;
- атрибут *directory* - каталог в который будет сгенерирована вся структура БД, представленная классами c#.

Подробнее о классах генерирующих код

Исходя из того, что каждый объект БД должен быть представлен сгенерированным классом, а также принимая во внимание, что для каждого типа объекта у нас должна быть своя сгенерированная структура код т.е. таблицы генерируются по своим правилам, а процедуры по своим, мы все же выделили общие черты для генераторов.

Интерфейс генератора содержит два метода:

- Generate – метод который будет генерировать код в определенное представления во время выполнения программы;
- WriteTo – метод который запишет код на языке C# в файл переданный в аргументы метода.

```
public interface IMetadataItemGenerator : IGenerator
{
   MetadataItem MetadataItem { get; }

   List<MetadataItem> GetChildren();
}
```

От данного интерфейса был унаследован интерфейс для генерации определённого элемента базы данных (свойство MetadataItem). А также добавлен метод, который возвращает дочерние элементы генерируемого объекта.

Для того, чтобы инструмент, с помощью которого мы будем генерировать код, мог пройтись по всем элементам БД и записать код куда укажет пользователь, он должен получить доступ к трем вещам:

- контейнер метаданных;
- конфигурация генератора;
- фабрика для создания генератора для определенного элемента.

```
    foreach (var item in
_container.Items.GetItemsRecursive<MetadataItem>(MetadataType.Namespaces))
    {
       var generator = _factory.GetItemGenerator(item, _config);
          generator.Generate().WriteTo(_config.GetOutputDirectory(item));
          if (!item.Default) continue;
          if (!item.Parent.Default)
              AddPublicStaticPropertyForItem(staticTablesClass, item);
          foreach (var child in generator.GetChildren())
          {
                   AddPublicStaticPropertyForItem(staticTablesClass, child);
                AssignPropertyInConstructor(constructor, item, child);
          }
    }
```

2.5 Выводы по разделу

В данном разделе был построен граф зависимостей интерфейсов и описана функциональность каждого из них. Приведена реализация интерфейсов SQL Clauses и интерфейсов для сгенерированных объектов БД. Подробно составлено руководство по использованию генерации кода и написанию запроса с использованием созданного API.

ПРОВЕРКА РАБОТОСПОСОБНОСТИ

2.6 Методика проверки

3.1.1 Unit – тесты

Для проверки работоспособности разработанного API для каждого модуля были написаны Unit – тесты, а также написаны тестовые реализации объектов БД [14].

Для этого был создан отдельный проект, который использует возможности UnitTestFramework. Каждый тест – отдельный метод, в названии которого отражено что мы проверяем, и что мы ожидаем от этой проверки.

Все методы помечаются атрибутом TestMethod и содержатся в классе с атрибутом TestClass, что позволяет MsTestAdapter запускать тесты прямо из VisualStudio.

При написании тестов был использована модель AAA.

Модель AAA (Arrange, Act, Assert) [15] – размещение, действие, утверждение является стандартным способом написания модульных тестов. Подраздел *Размещение* инициализирует объекты и устанавливает значения данных, которые нужны методу для теста. Подраздел *Действие* вызывает методы для теста с размещенными параметрами Подраздел *Утверждение* проверяет, чтобы метод для теста действовал, как ожидается.

```
    [TestMethod]
    public void DerivedTable_GetFieldByName_ShouldReturnExistingField()
    {
       /// Arrange
          var derivedTable = _selectStep.Select(Tables.Orders.OrderId,
SqlExpression.Value("column2"), SqlExpression.Value(14)).AsTable("query1");
          /// Act
          var fc = derivedTable.Field("OrderID");
          /// Assert
          Assert.IsNotNull(fc);
          Assert.AreEqual(Tables.Orders.OrderId.FieldName, fc.FieldName);
    }
```

В данном тесте мы проверяем класс DerivedTable метод GetFieldByName и ожидаем, что после создания запроса с выбранным полем из таблицы Orders мы сможем получить это поле и оно не будет равно NULL, а также оно должно быть эквивалентно тому полю, которое мы передали в метод Select.

В данном методе проверяются общие возможности созданного API, а именно каждый метод связанный с SQL Clause. В подразделе *размещение* мы создаем

контекст, в котором уже настроена схема БД и выбран MS SQL диалект, все эти настройки мы задаем в методе Init тестируемого класса.

```
    [TestMethod]
    public void CommonApiTest()
    {
       /// Arrange
       var ctx = DSL.SqlContext;
          var query = ctx.GetNewQueryDSL();
          /// Act
          var sql = query
                   .Select(SqlExpression.Value(5), Tables.Orders.OrderId.As("id"),
Tables.Customers.CustomerId, SqlExpression.Count(Tables.Orders.OrderId))
                    .From(Tables.Orders)
                        .InnerJoin(Tables.Customers)
                            .On(Tables.Orders.OrderId ==
Tables.Customers.CustomerId)
                    .Where(Tables.Orders.OrderId.Greater(10))
                        .And("1 < 2")
                    .Having(Tables.Orders.OrderId.Count().Greater(5))
                    .OrderBy(Tables.Orders.OrderId)
                    .SQL;
          /// Assert
         Assert.AreEqual("Select 5, \"Orders\".\"OrderID\" id,
       \"Customers\".\"CustomerID\", Count(Orders.OrderID) " +
                                "From \"Orders\" " +
                                    "Inner Join \"Customers\" " +
                                        "On Orders.OrderID = Customers.CustomerID "
+
                                "Where (Orders.OrderID > 10) And 1 < 2 " +
                                "Having (Count(Orders.OrderID) > 5) " +
                                "Order By Orders.OrderID", sql, true);
            }
```

Метод GetNewQueryDSL возвращает нам объект с помощью, которого мы создаем запрос и, собственно, его мы и тестируем. В подразделе *утверждение* мы приводим строку ожидаемого запроса и сравниваем ее со свойством SQL объекта DSL.

Тесты для полей таблиц

Основные сценарии использования полей – это добавление поля в списки выбора, группировки или сортировки. Поэтому уделим особое внимание проверке этих методов и убедимся, что элементы добавляются корректно. Для теста было объявлено поле класса TableField. В каждом тесте мы создаем поле и соответствующий метод для добавления в список. Как говорилось в пункте 2.3, метод добавления в список принимает на вход сервис, который содержит низкоуровневую логику и объектную модель запроса, в данном случае это объект

типа UnionSubQuery т.е. подзапрос, который содержится в группе, объединенной оператором UNION или UNION ALL.

```
    [TestMethod]
    public void FieldWithoutAlias_AddToSelectList_shouldAddSelectItem()
    {
       /// Arrange
       OrderID = TableField<int>.Create("OrderID", _orders, _ordersMetadataObject);
          var firstSelect = _queryRoot.FirstSelect();
       /// Act
          OrderID.AddToSelectList(_queryModifier, firstSelect);
       /// Assert
          Assert.AreEqual(1, firstSelect.SelectList.Count);
          Assert.AreEqual("\"Orders\".\"OrderID\"", firstSelect.SelectListString,
true);
```

Чтобы создать любое поле, принадлежащее таблице, нам нужно в конструктор класса TableField передать название поля, ссылку на таблицу, которая реализует интерфейс ITable и MetadataObject – объект, который описывает метаданные таблицы и хранится в контейнере метаданных, который заполняется при присвоении SQL контексту схемы БД. После вызова метода добавления в список выбранных полей мы проверяем, что в объектной модели, а конкретно в списке выбора, содержится один элемент, и он принадлежит таблице Orders, а его имя – OrderID.

Так же один из наиболее вероятных сценариев использования поля таблицы – это использование его с присвоением ему псевдонима. Для этого тоже был создан отдельный тестовый метод.

```
    [TestMethod]
    public void Field_As_shouldReturn_newFieldWithAlias()
    {
       /// Arrange
       OrderID = TableField<int>.Create("OrderID", _orders,
_ordersMetadataObject);
       /// Act
          var aliasedField = OrderID.As("id");
       /// Assert
          Assert.IsNotNull(aliasedField);
          Assert.AreNotEqual(OrderID, aliasedField);
          Assert.AreEqual("id", aliasedField.Alias, true);
    }
```

Из названия видно, что мы ожидаем получить новое поле, которое будет содержать в себе Alias переданный в метод As. Для этого в подразделе

утверждение мы проверяем что ссылка поля OrderID не равна новому полю aliasedField.

Тесты для таблиц

Основная роль таблиц в запросе — это быть источником данных т.е. контейнером полей, которые содержат в себе значения. Основной метод, который используется для добавления таблицы в запрос, предоставляет интерфейс IFromClauseItem. Поэтому будем использовать тестовую реализацию таблицы Orders.

```
[TestMethod]
public void TableWithAlias_AddToFromClause_ShouldAddOrdersDataSource()
{
   /// Arrange
   _orders = new Orders(_ordersParentList){Alias = "o"};
      var firstSelect = _queryRoot.FirstSelect();
   /// Act
      _orders.AddToFromClause(_queryModifier, firstSelect);
   /// Assert
      Assert.AreEqual(1, firstSelect.FromClause.Count);
      Assert.AreEqual("\"Orders\" o", firstSelect.FromClauseString, true);
}
```

В данном тестовом методе критерием того, что тест прошел проверку, является то, что после вызова метода AddToFromClause в объектную модель запроса добавится новый источник данных и он будет единственным, при этом имя и псевдоним должны совпадать с ожидаемыми именами т.е. имя – это «Orders», а псевдоним – «o».

Поля таблиц инициализируются в конструкторе наследника базового класса Table<T>. Поэтому, во-первых, стоит проверить вызывается ли этот метод при создании объекта таблицы, а во-вторых, правильно ли он инициализирует поля.

```
[TestMethod]
public void Table_PropertyOrderDate_shouldReturn_NotNullField()
{
   _orders = new Orders(_ordersParentList);
      var field = _orders.OrderDate;
      Assert.IsNotNull(field);
      Assert.IsInstanceOfType(field, typeof(TableField<DateTime>));
}
```

Для того чтобы создать таблицу нам нужно в конструктор передать ссылку на список таблиц из контейнера метаданных, который инициализируется в текущем экземпляре SQL контекста. В подразделе *утверждение* происходит проверка, что

полученное поле из экземпляра класса Orders не является NULL и поле является экземпляром класса TableField от типа DateTime.

Тесты для подпрограмм

Некоторые подпрограммы можно использовать как источники данных т.е. их можно передать в метод From (например, функции, возвращающие табличное значение), некоторые можно использовать в методе Select (например, скалярные функции).

Все сгенерированные функции хранятся в статическом классе Routines, откуда их можно вызвать как обычный метод, передав соответствующие параметры (если они имеются у подпрограммы).

Реализуем тесты на добавление подпрограммы в список выбора и список источников данных.

```
    [TestMethod]
    public void Routine_AddToSelectList_Test()
    {
       /// Arrange
       var proc = Routines.GetSomething(SqlExpression.Value(1),
                                        SqlExpression.Value("nameofStor"));
          var queryRoot = new QueryRoot(DSL.SqlContext);
          var firstSelect = queryRoot.FirstSelect();
       /// Act
          proc.AddToSelectList(SQLQueryService.Instance, firstSelect);
       /// Assert
          Assert.AreEqual(1, firstSelect.SelectList.Count);
          Assert.AreEqual("Select getSomething(1, 'nameofStor') From",
firstSelect.SQL);
    }
```

В данном тестовом методе мы используем тестовую реализацию сгенерированной процедуры, которая принимает 2 параметра. При вызове метода GetSomething, название которого эквивалентно названию процедуры, мы получаем объект класса GetSomething. В подразделе *размещение* мы создаем объектную модель запроса QueryRoot, инициализируя ее текущим SQL контекстом, где уже содержится контейнер с метаданными БД. И в подразделе утверждение мы проверяем, что в список выбора добавилось выражение вызова процедуры с переданными параметрами.

Тесты для генерации кода

Для того чтобы протестировать каждый класс-генератор можно выделить абстрактные критерии для проверки, которые будут универсальными для каждого из генератора. Для этого создадим базовый абстрактный класс, в котором будут

содержатся абстрактные методы, которые будут реализованы уже конкретным классом для проверки определенного генератора.

Список абстрактных функций.

Initialize – функция вызываемая при создании класса наследника, инициализирующая необходимые поля помеченные в базовом классе как protected.

ExpectedAttributeName – функция возвращающая значение свойства Name класса NameAttribute. Так как генерируемые классы пространств имен, таблиц и представлений содержат атрибут, который хранит в себе имя, назначенное в БД, мы должны обязательно проверить совпадает ли он с ожидаемым именем в БД.

ExpectedBaseClassName – функция возвращающая имя типа от которого наследован генерируемый класс. То есть если мы создаем тестовый класс для проверки генератора пространств имен, эта функция должна нам вернуть строку вида «AbstractNameSpace», так как все пространства имен должны быть унаследованы именно от этого класса, если же проверяются генераторы для таблиц, то данная функция вернет – «Table».

ExpectedNumberOfProperties – функция возвращает количество сгенерированных свойств для таблиц соответсвенно это количество полей, для представлений количество возвращаемых колонок.

ExpectedChildren – функция возвращающая список дочерних элементов генерируемой структуры. Для таблиц и предствалений это соответсвенно поля, а для пространств имен соответсвенно в зависимости от иерархии схемы, пакеты, таблицы, подпрограммы и т.д.

Общие критерии проверки

```
[TestMethod]
public void CodeUnit_Should_Contains_CoreAndFluentImportNamespaces()
{
      Generator.Generate();
      var codeUnit = GetCodeUnit();
      var imports = (from CodeNamespaceImport import in
codeUnit.Namespaces[0].Imports select import.Namespace).ToList();

      Assert.IsTrue(imports.Contains("ActiveQueryBuilder.Core"));
      Assert.IsTrue(imports.Contains("ActiveQueryBuilder.Core.FluentSql"));
}
```

Данный тестовый метод извлекает из генератора DOM, и проверяет наличие подключенных пространств имен «ActiveQueryBuilder.Core» в котором содержится реализованная логика по построению объектной модели запроса и «ActiveQueryBuilder.Core.FluentSQL» пространство имен, содержащее в себе реализованный API по построению запроса.

```
[TestMethod]
public void CodeUnit_Should_ContainsType_WithNameProvidedByConfig()
{
      Generator.Generate();
      var codeUnit = GetCodeUnit();
      var namespaces = (from CodeNamespace codeNamespace in codeUnit.Namespaces
select codeNamespace).ToList();

      var targetNs = namespaces.First(ns => ns.Name == Configuration.Namespace);

      Assert.AreEqual(Configuration.GetClassName(Metadata),
targetNs.Types[0].Name);
}
```

Данный тестовый метод извлекает из генератора DOM, и проверяет совпадает ли название пространства имен для генерируемых классов, указанное в конфигурации.

```
[TestMethod]
public void CodeUnit_TargetClass_InheritsFromBaseType()
{
      Generator.Generate();
      var codeUnit = GetCodeUnit();
      var targetType = GetTargetClass(codeUnit);
      Assert.AreEqual(ExpectedBaseClassName(), targetType.BaseTypes[0].BaseType);
}
```

Данный тестовый метод извлекает из генератора DOM, и проверяет что сгенерированный класс унаследован от ожидаемого базового класса. То есть если мы создаем тестовый класс для проверки генератора пространств имен, эта функция должна нам вернуть строку вида «AbstractNameSpace», так как все пространства имен должны быть унаследованы именно от этого класса, если же проверяются генераторы для таблиц, то данная функция вернет – «Table».

```
[TestMethod]
public void Generator_ChildrenTest()
{
      CollectionAssert.AreEqual(ExpectedChildren(), Generator.GetChildren());
}
```

Данный метод проверяет, что генератор возвращает правильный список дочерних элементов.

Реализация тест классов для генераторов

Для таблиц:

```
    [TestClass]
    public class TableGenTests : GeneratorBaseTests
    {
        [TestInitialize]
        public override void Initialize()
        {
            Metadata =
Ctx.MetadataContainer.FindItem<MetadataObject>("\"Orders\"");
            Generator = new MetadataObjectGenerator((MetadataObject)Metadata,
Configuration);
        }

        protected override string ExpectedAttributeName()
        {
            return typeof(NameAttribute).FullName;
        }

        protected override string ExpectedBaseClassName()
        {
            return "ActiveQueryBuilder.Core.FluentSql.Table`1";
        }

        protected override int ExpectedNumberOfProperties()
        {
            return Metadata.Items.Fields.Count;
        }

        protected override List<MetadataItem> ExpectedChildren()
        {
            return new List<MetadataItem>();
        }
    }
```

В данном классе остается только переопределить абстрактные фукнции определенные в классе GeneratorBaseTests описание функционала которых представлено выше. Здесь в качестве проверяемого элемента БД взята таблица Orders из схемы Northwind.

Для пространств имен:

```
    [TestClass]
    public class NamespaceGenTests : GeneratorBaseTests
    {
        [TestInitialize]
        public override void Initialize()
        {
            Metadata =
Ctx.MetadataContainer.Items.FindItem<MetadataNamespace>("Northwind",
MetadataType.Namespaces);
            Generator = new MetadataNamespaceGenerator(new GeneratorsFactory(),
(MetadataNamespace)Metadata, Configuration, MetadataType.Namespaces);
        }
protected override string ExpectedAttributeName(){return
typeof(NameAttribute).FullName;}
        protected override string ExpectedBaseClassName()
        {
            return "ActiveQueryBuilder.Core.FluentSql.AbstractNamespace";
        }
    }
```

В данном классе остается только переопределить абстрактные фукнции определенные в классе GeneratorBaseTests описание функционала которых представлено выше. В качестве проверяемого элемента БД взята вся схема БД под названием Northwind.

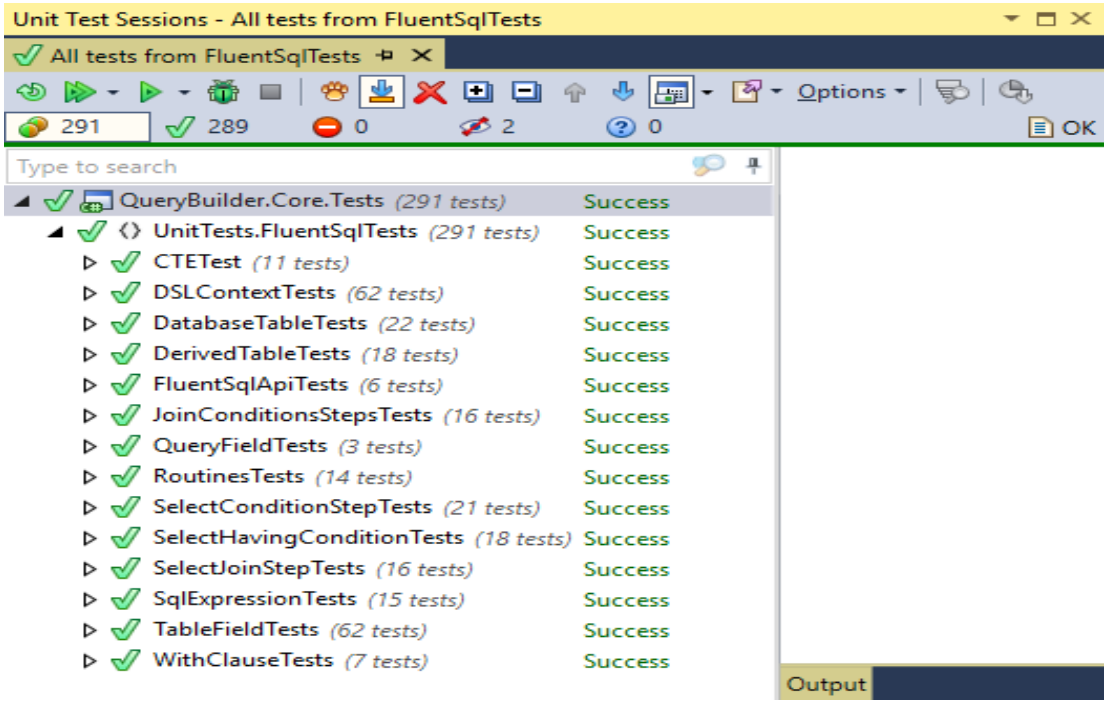

Рисунок 3.1 – Запуск и проверка тестов в Unit testing framework

3.1.2 Реализация Demo – проекта

Следуя инструкциям (см. приложение 1), была сгенерирована схема БД под названием Sakila.

Данная БД содержит в себе одну схему с таблицами и представлениями. После генерации в папке, с названием указанном в конфигурационном файле, создаются каталоги для схем, таблиц и представлений (каталоги не создаются для тех объектов, которые не присутствуют в БД). Так же в главный каталог добавляются 3 статических класса:

- DSL – содержит свойство Query, которое по сути является основным источником функциональности для построения запроса;
- Tables – содержит все пространства имен и объекты которые содержатся в них, а так же имеет в качестве свойств объекты по умолчанию указанные в схеме БД;
- Routines – содержит подпрограммы.

Сгенерированная структура БД в каталоге проекта:

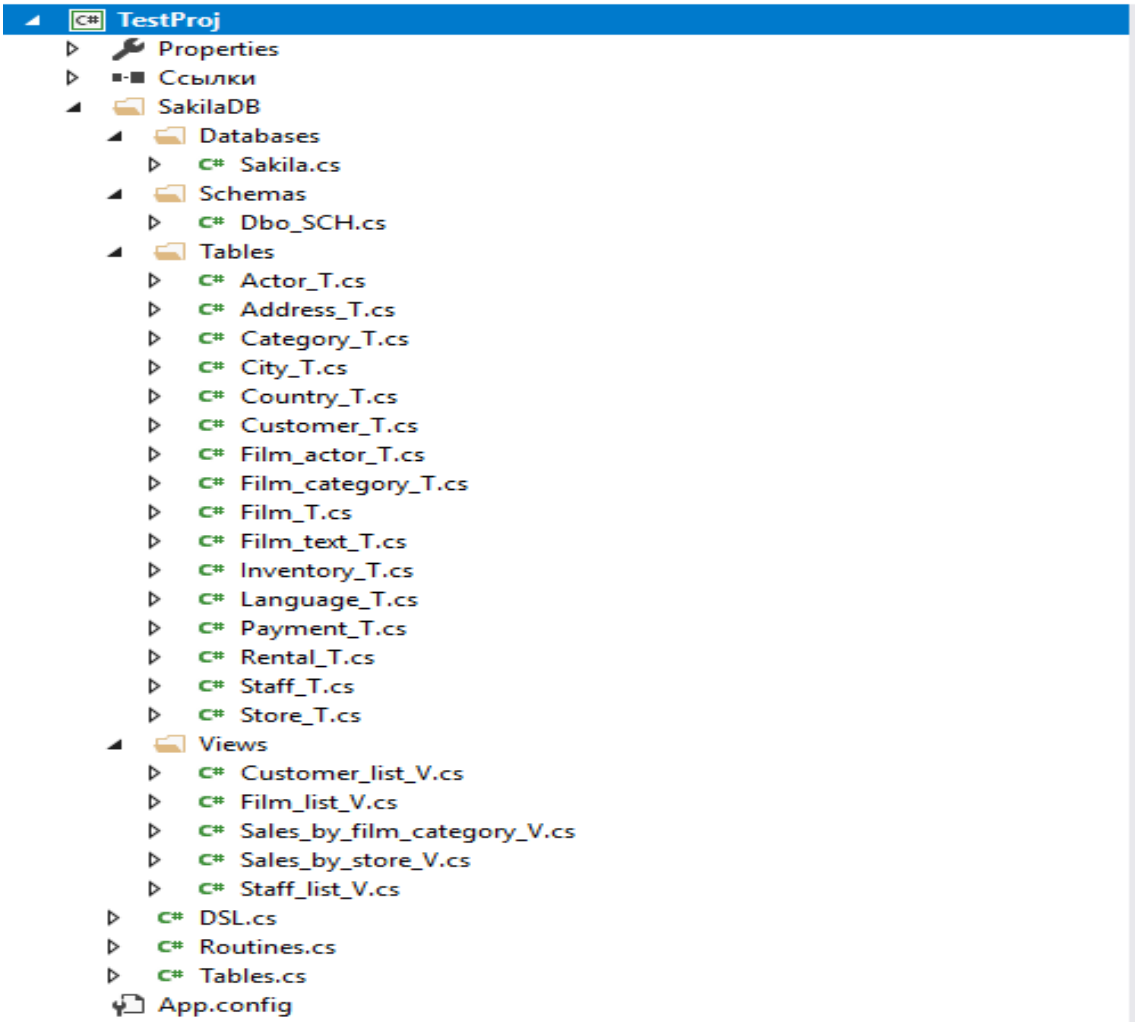

Рисунок 3.1 – Сгенерированная структура БД

В качестве примеров были написаны небольшие запросы с использованием сгенерированных классов.

```
    public static string Select_CommonStatement()
    {
       var result = DSL.Query
                      .Select(Actor.First_name, Actor.Last_name, Count())
                .From(Actor, Film_actor)
                .Join(Film).On(Film.Film_id ==
 Film_actor.Film_id).And(Actor.Actor_id == Film_actor.Actor_id)
                .Where(Film.Length.Greater(95))
                         .And(Film.Rating.Equal("good"))
                .GroupBy(Actor.First_name, Actor.Last_name)
                .Having(Count().Greater(5))
                .OrderBy(Actor.Last_name.Asc())
                .Limit(2)
                .Offset(1)
                .Result;

                return Program.Formatted(result);
    }
```

Данный метод показывает основную функциональность API с использованием сгенерированных объектов и всех главных SQL clasuses.

```
D:\aqb\_deployment\Files\Samples\cs\FluentSqlDemo\FluentSqlDemo\bin\Debug\FluentSqlDemo.exe
SELECT actor.first_name,
  actor.last_name,
  Count(*)
FROM actor,
  film_actor
  INNER JOIN film ON film.film_id = film_actor.film_id AND actor.actor_id =
    film_actor.actor_id
WHERE (film.length > 95) AND film.rating = 'good'
GROUP BY actor.first_name,
  actor.last_name
HAVING (Count(*) > 5)
ORDER BY actor.last_name
```

Рисунок 3.2 – Результат выполнения метода Select_CommonStatement()

2.7 Выводы по разделу

В данном разделе была проверена работоспособность созданного API. Для этого для каждого функционального модуля с помощью модели AAA были написаны Unit – тесты. Так же на тестах проверена работоспособность инструмента для генерации кода из структуры метаданных БД.

С помощью реализованного генератора метаданных БД была сгенерирована тестовая схема и создан проект в Visual Studio для демонстрации работоспособности всей разработанной системы.

ЗАКЛЮЧЕНИЕ

В данной работе разработана библиотека классов для преобразования запросов SQL на языке C#, которая в полной мере отражает основные функциональные возможности языка SQL и позволяет генерировать структуру БД в классы языка c#.

Проведен анализ существующих решений, выделены общие черты в программных интерфейсах, которые позволили правильно построить архитектуру ПО. В пункте 1.3 были определены основные требования к системе.

Также был построен граф зависимостей интерфейсов и описана функциональность каждого из них. Приведена реализация интерфейсов SQL Clauses и интерфейсов для сгенерированных объектов БД. Подробно составлено руководство по использованию генерации кода и написанию запроса с использованием созданного API.

В конечном итоге была проверена работоспособность созданного API. Для этого для каждого функционального модуля с помощью модели AAA были написаны Unit – тесты. Так же на тестах проверена работоспособность инструмента для генерации кода из структуры метаданных БД.

С помощью реализованного генератора метаданных БД была сгенерирована тестовая схема и создан проект в Visual Studio для демнострации работоспособности всей разработанной системы.

В дальнейшем планируется добавить возможность из SQL скрипта генерировать код создания запроса, возможность представлять данные результата SQL-запроса в объектном виде, а также добавить новые возможности для упрощения написания запроса.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

. Купер, А. Психбольница в руках пациентов / А. Купер, Р. Рейман, Д.Кронин. – СПБ. Символ-Плюс, 2009. – 688 с.

. Рихтер Д. CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C#. – 4-е изд. Питер, 2017.– 333-356 с.

. Мартин Р.К. Чистая архитектура. Искусство разработки программного обеспечения. Питер, 2018. 192 с.

. Мартин Р.К. Идеальный программист. Как стать профессионалом разработки ПО. – Питер, 2017. – 42-56 с.

. Мартин Ф. UML. Основы. Краткое руководство по стандартному языку объектного моделирования.– 3-е изд. Символ-Плюс, 2006. – 125 с.

. Алан Б. Изучаем SQL. – Символ-Плюс, 2016. – 90-110 с.

. Грофф Д. SQL. Полное руководство. / Вайнберг П., Оппель Э. – 3-е изд. Вильямс, 2014. – 687-700 с.

. Тарасов, С.В. СУБД для программиста. Базы данных изнутри. / С.В. Тарасов. – М.: СОЛОН-Пресс, 2015. – 320 с.

. Мартин Р.К. Чистый код. Создание, анализ и рефакторинг. – Питер, 2010. – 253 с.

0. Джон С. C# для профессионалов. Тонкости программирования. – Вильямс, 2017. – 420-500 с.

1. Сергей Т. Паттерны проектирования на платформе NET. – Питер, 2016.– 137-145 с.

2. Джозеф А.. C# 5.0. Справочник. Полное описание языка. / Албахари Б.– 5-е изд. Вильямс, 2014.– 763-790 с.

3. Эндрю Т. Язык программирования C# 5.0 и платформа.NET 4.5. – 6-е изд. Вильямс, 2015.– 700-800 с.

4. Майерс, Г. Д. Искусство тестирования программ / Пер. с англ. под ред. Б. А. Позина / Г. Д. Майерс. – М. : Финансы и статистика, 1982. – 176 с

5. Рой О. Искусство автономного тестирования с примерами на C#. –2-е изд. ДМК Пресс, 2014. – 61-70 с.

ПРИЛОЖЕНИЕ 1

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

1. Назначение программы

Данная система предназначена для эффективного написания запросов SQL на объектно-ориентированном языке программирования C#. Система представляет собой DSL (Domain specific language), который по синтаксису приближен к SQL, вместе с системой прилагается генератор схемы БД в классы языка C#.

2. Условия выполнения программы

Для работы системы необходим компьютер с объемом оперативной памяти не менее 1Гб оперативной памяти, тактовой частотой процессора не менее 2 ГГц. Библиотека ActiveQueryBuilder.dll содержащая реализацию описываемой системы. Установленная СУБД и подключение к ней.

3. Выполнение программы

Для того чтобы сгенерировать существующую схему БД необходимо настроить конфигурационный файл.

```
    <configuration>
        <configSections>
            <section name="generation"
type="ActiveQueryBuilder.Core.MetadataCodeGeneration.GeneratorConfig,
ActiveQueryBuilder.Core"/>
        </configSections>
        <generation>
            <syntaxProvider type="ActiveQueryBuilder.Core.MSSQLSyntaxProvider,
ActiveQueryBuilder.Core" />
            <metadataProvider type="ActiveQueryBuilder.Core.MSSQLMetadataProvider,
ActiveQueryBuilder.MSSQLMetadataProvider"/>
            <metadataSource xml="D:\tmp\NorthwindWithFuncs.xml">
                <dbConnection type="System.Data.SqlClient.SqlConnection,
System.Data, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
connectionString="Data Source=MY-PC\SQLEXPRESS;Initial Catalog=db_main;Integrated
Security=True" />
            </metadataSource>
            <target namespace="Sakila"
directory="D:\tmp\TestProj\TestProj\SakilaDB"/>
        </generation>
```

Секция <configSections>:

Регистрация секции конфигурации.

Определяет, каким классом (атрибут type) будет представлена конфигурация в коде проекта. Атрибут name = ’Имя_зарегестрированной_секции’, имя может быть любым.

Секция <’Имя_зарегестрированной_секции’>:

Обязательный тег <*syntaxProvider*>, определяющий SQL диалект, в атрибуте type указываем полное имя сборки соответствующей библиотеки.

Тег <*metadataSource*>*:*

– Атрибут *xml* - абсолютный путь к вашему файлу метаданных.

Тег <*dbConnection*> - конфигурация соединения с базой данных в реальном времени.

Атрибут type – полное имя класса для установки соединения.

Атрибут *connectionString* – строка подключения к вашей БД.

Замечания: если присутствует тег *dbConnection*, то атрибут *xml* нужно удалить.

Тег <*target*>:

Атрибут *namespace* – имя пространства имен, в котором будут находиться все сгенерированные классы.

Атрибут *directory* - каталог в который будет сгенерирована вся структура БД, представленная классами c#.

Генерация кода:

- В коде проекта:

```
namespace TestProj
{
    class Program
    {
        static void Main(string[] args)
        {
            var configuration =
MetadataGenerationConfiguration.Create("generation", @"sakila.config");
            var genTool = new GenerationTool(configuration);
            genTool.Generate();
        }
    }
}
```

- Через командную строку

Из каталога, где находится generation-tool.exe, выполните команду: Generation-tool.exe “Имя_секции” “Абсолютный_путь_до_файла_конфигурации”

Замечания:

В каталоге должна находиться сборка ActiveQueryBuilder.core.

В случае если в конфигурации вы используете сборки не входящие в ActiveQueryBuilder.core или System.Data, положите их в каталог, где находится Generation-tool.exe.

Пример использования сгенерированного кода в проекте:

```
class Program
{
    static void Main(string[] args)
    {
        var result = DSL.Query
            .Select(Actor.First_name, Actor.Last_name, Count())
            .From(Actor, Film_actor)
                .Join(Film).On(Film.Film_id ==
Film_actor.Film_id).And(Actor.Actor_id == Film_actor.Actor_id)
            .Where(Film.Length.Greater(95))
                .And(Film.Rating.Equal("good"))
            .GroupBy(Actor.First_name, Actor.Last_name)
            .Having(Count().Greater(5))
            .OrderBy(Actor.Last_name.Asc())
            .Limit(2)
            .Offset(1)
            .Result;
        return Program.Formatted(result);
    }
}
```

SQL:

```
SELECT actor.first_name, actor.last_name, Count(*)
FROM actor, film_actor
    INNER JOIN film
        ON film_actor.actor_id = film_actor.actor_id AND
            film.film_id = film.film_id
WHERE (film.length > 95) AND film.rating = 'good'
GROUP BY actor.first_name, actor.last_name
HAVING (Count(*) > 5)
ORDER BY actor.last_name
OFFSET 1 ROWS
FETCH FIRST 2 ROWS ONLY
```

ТЕКСТ ПРОГРАММЫ

Файл DSLContext.cs

```
public class DSLContext : IDSLContext, IDisposable
    {
        private readonly IQueryModifier _queryModifier;
        private readonly UnionSubQuery _activeUsq;
        private readonly SQLContext _sqlContext;
        private readonly QueryRoot _queryRoot;
        private readonly List<Action> _beforeGetResultConditionsActions = new
List<Action>();
        private readonly HashSet<DataSourceBase> _autoAddedDataSources = new
HashSet<DataSourceBase>();
        private QueryStatistics _queryStatistics;
        private string _asDerivedTableAlias;
        private string _asSelectItemAlias;

        public StatisticsDatabaseObjectList UsedDatabaseObjects
        {
            get
            {
                if (_queryStatistics == null)
                    _queryStatistics =
QueryStatisticsService.GetStatistics(_queryRoot);

                return _queryStatistics.UsedDatabaseObjects;
            }
        }
        public StatisticsFieldList UsedDatabaseObjectFields
        {
            get
            {
                if (_queryStatistics == null)
                    _queryStatistics =
QueryStatisticsService.GetStatistics(_queryRoot);

                return _queryStatistics.UsedDatabaseObjectFields;
            }
        }
        public StatisticsOutputColumnList OutputColumns
        {
            get
            {
                if (_queryStatistics == null)
                    _queryStatistics =
QueryStatisticsService.GetStatistics(_queryRoot);
```

```
                return _queryStatistics.OutputColumns;
            }
        }
        public List<IField> Fields
        {
            get
            {
                if (string.IsNullOrEmpty(_asDerivedTableAlias))
                    _asDerivedTableAlias = CreateUniqueAlias(_queryRoot,
"query_");
                return AllFields();
            }
        }
        public QueryRoot Result
        {
            get
            {
                BeforeGetResult();
                return _queryRoot;
            }
        }
        public IQueryModifier QueryModifier { get { return _queryModifier; } }
        public UnionSubQuery ActiveUnionSubQuery { get { return _activeUsq; } }
        public string SQL { get { return Result.SQL; }  }

        public DSLContext(SQLContext sqlContext)
        {
            _queryRoot = new QueryRoot(sqlContext);
            _sqlContext = sqlContext;
            _activeUsq = _queryRoot.FirstSelect();
            _queryModifier = SQLQueryService.Instance;
            _queryRoot.Updated += _queryRoot_Updated;
        }
        private DSLContext(QueryRoot queryRoot, UnionSubQuery newUnionSubQuery)
        {
            _queryRoot = queryRoot;
            _sqlContext = queryRoot.SQLContext;
            _activeUsq = newUnionSubQuery;
            _queryModifier = SQLQueryService.Instance;
            _queryRoot.Updated += _queryRoot_Updated;
        }
        private void _queryRoot_Updated(object sender, EventArgs e)
        {
            if (_queryStatistics != null)
                _queryStatistics.ReloadStatistics(new
AstStatistics(_queryRoot.ResultAST));
        }
        public ISelectResult As(string alias)
        {
```

```
            _asSelectItemAlias = alias;
            return this;
        }
        public IWithAsStep With(string alias)
        {
            return new WithAsStep(this, alias);
        }
        public IDSLContext With(ICommonTableExpression cte)
        {
            using (new UpdateRegion(_queryRoot))
            {
                var withClause = _queryRoot.AddNewCTE(cte.SubQuery, cte.CteName);
                if (cte.Columns == null) return this;
                if (withClause.ColumnNamesList == null)
                {
                    withClause.ColumnNamesList = new
AstAliasExpressionsList(_sqlContext);
                }
                foreach (var column in cte.Columns)
                {
                    AstToken token;
                    if (!string.IsNullOrEmpty(column) && column.StartsWith("'"))
                        token = new AstTokenString(_sqlContext, column);
                    else
                        token = _sqlContext.ParseIdentifier(column);
                    var sqlAlias = new SQLAliasExpression(_sqlContext)
                    {
                        Alias = token
                    };
                    withClause.ColumnNamesList.Add(sqlAlias);
                }
            }
            return this;
        }
        public ISelectOrderByStep Union(Action<ISelectStep> selectAction)
        {
            var newUsq = _activeUsq.ParentGroup.Add();
            var innerSelect = new DSLContext(Result, newUsq);
            selectAction(innerSelect);
            innerSelect.BeforeGetResult();
            using (var stat = QueryStatisticsService.GetStatistics(newUsq))
            {
                if (!IsCorrectUnionStep(stat))
                    throw new Exception("Incorrect union");
            }
            return this;
        }
        public ISelectOrderByStep UnionAll(Action<ISelectStep> selectAction)
        {
```

```
        var newUsq = _activeUsq.ParentGroup.Add();
        newUsq.UnionAllFlag = true;
        var innerSelect = new DSLContext(Result, newUsq);
        selectAction(innerSelect);
        innerSelect.BeforeGetResult();
        using (var stat = QueryStatisticsService.GetStatistics(newUsq))
        {
            if (!IsCorrectUnionStep(stat))
                throw new Exception("Incorrect union");
        }
        return this;
    }
    public ISelectOrderByStep Union(ISelectResult select)
    {
        if (!IsCorrectUnionStep(select))
            throw new Exception("Incorrect union");
        var newUsq = _activeUsq.ParentGroup.Add();
        newUsq.SQL = select.SQL;
        return this;
    }
    public ISelectOrderByStep UnionAll(ISelectResult select)
    {
        if (!IsCorrectUnionStep(select))
            throw new Exception("Incorrect union");
        var newUsq = _activeUsq.ParentGroup.Add();
        newUsq.UnionAllFlag = true;
        newUsq.SQL = select.SQL;
        return this;
    }
    public ISelectOffsetStep Limit(int val)
    {
        return Limit(val.ToString());
    }
    public ISelectOffsetStep Limit(string limit)
    {
        _activeUsq.ParentSubQuery.LimitCount = limit;
        return new SelectOffsetStep(this);
    }
    public ISelectAfterLimitOffsetStep Offset(int val)
    {
        return Offset(val.ToString());
    }
    public ISelectAfterLimitOffsetStep Offset(string offset)
    {
        _activeUsq.ParentSubQuery.LimitOffset = offset;
        return new SelectAfterLimitOffsetStep(this);
    }
    public ISelectLimitStep OrderBy(params string[] orderByItems)
    {
```

```
                if (orderByItems == null)
                    throw new ArgumentNullException("orderByItems");
                foreach (var orderByItem in orderByItems)
                {
                    var expressionAst = _sqlContext.ParseOrderByItem(orderByItem);
                    _queryModifier.AddOrderByItem(_activeUsq, new
OrderByItem(expressionAst, _sqlContext));
                }
                return this;
            }
            public ISelectLimitStep OrderBy(params IOrderByItem[] orderByItems)
            {
                if (orderByItems == null)
                    throw new ArgumentNullException("orderByItems");
                foreach (var orderByItem in orderByItems)
                {
                    orderByItem.AddToOrderByList(QueryModifier, ActiveUnionSubQuery);
                }
                return this;
            }
            public ISelectHavingConditionStep Having(ConditionBase condition)
            {
                if (condition == null)
                    throw new ArgumentNullException("condition");
                var conditionsList = new ConditionsList(_sqlContext) {condition};
                var conditionStep =  new SelectHavingConditionStep(this,
ActiveUnionSubQuery.HavingConditions, conditionsList);

_beforeGetResultConditionsActions.Add(conditionStep.AddConditionsToParent);
                return conditionStep;
            }
            public ISelectHavingConditionStep Having(Condition condition)
            {
                return Having(condition.ExpressionStr);
            }
            public ISelectHavingConditionStep Having(string condition)
            {
                if (string.IsNullOrEmpty(condition))
                    throw new ArgumentNullException("condition");
                var sqlExpressionItem = _sqlContext.ParseLogicalExpression(condition);
                var conditionItem = new ConditionItem(sqlExpressionItem, _sqlContext);
                return Having(conditionItem);
            }
            public ISelectHavingStep GroupBy(params string[] aggregatedItems)
            {
                if (aggregatedItems == null)
                    throw new ArgumentNullException("aggregatedItems");
                foreach (var aggregatedItem in aggregatedItems)
                {
```

```
                    var expressionAst = _sqlContext.ParseExpression(aggregatedItem);
                    var groupingItem = new GroupingItem(expressionAst,
_activeUsq.FromColumns, _sqlContext);
                    SQLQueryService.Instance.AddGroupingItem(_activeUsq,
groupingItem);
                }
                return this;
            }
            public ISelectHavingStep GroupBy(params IGroupingItem[] groupingItems)
            {
                if (groupingItems == null)
                    throw new ArgumentNullException("groupingItems");
                foreach (var groupingItem in groupingItems)
                {
                    groupingItem.AddToGroupByList(_queryModifier,
ActiveUnionSubQuery);
                }
                return this;
            }
            public ISelectConditionStep Where(string condition)
            {
                if (string.IsNullOrEmpty(condition))
                    throw new ArgumentNullException("condition");
                var expressionAst = _sqlContext.ParseLogicalExpression(condition);
                return Where(new ConditionItem(expressionAst, _sqlContext));
            }
            public ISelectConditionStep Where(Condition condition)
            {
                return Where(condition.ExpressionStr);
            }
            public ISelectConditionStep Where(ConditionBase condition)
            {
                if (condition == null)
                    throw new ArgumentNullException("condition");
                var conditionsList = new ConditionsList(_sqlContext){condition};
                var conditionStep = new SelectConditionStep(this,
_activeUsq.WhereConditions, conditionsList);

_beforeGetResultConditionsActions.Add(conditionStep.AddConditionsToParent);
                return conditionStep;
            }
            public ISelectConditionStep WhereExists(Action<ISelectStep> selectAction)
            {
                using (var select = new DSLContext(_sqlContext))
                {
                    selectAction(@select);
                    return Where("Exists(" + @select.SQL + ")");
                }
            }
```

```
            public ISelectConditionStep WhereExists(ISelectResult @select)
            {
                return Where("Exists(" + select.SQL + ")");
            }
            public ISelectConditionStep WhereNotExists(Action<ISelectStep>
selectAction)
            {
                using (var select = new DSLContext(_sqlContext))
                {
                    selectAction(@select);

                    return Where("Not Exists(" + @select.SQL + ")");
                }
            }
            public ISelectConditionStep WhereNotExists(ISelectResult @select)
            {
                return Where("Not Exists(" + select.SQL + ")");
            }
            public ISelectJoinStep From(params string[] dataSources)
            {
                if (dataSources == null)
                    throw new ArgumentNullException("dataSources");
                for (int i = 0; i < dataSources.Length - 1; i++)
                {
                    SQLQueryService.Instance.AddDataSourceFromExpression(_activeUsq,
dataSources[i]);
                }
                var lastDataSource =
SQLQueryService.Instance.AddDataSourceFromExpression(_activeUsq, dataSources.Last());
                return new SelectJoinStep(this, lastDataSource);
            }
            public ISelectJoinStep From(params IName[] names)
            {
                return From(names.Select(name => name.Name).ToArray());
            }
            public ISelectJoinStep From(params IFromClauseItem[] items)
            {
                if (items == null)
                    throw new ArgumentNullException("items");
                DataSource last = null;
                for (int i = 0; i < items.Length; i++)
                {
                    var currentDataSource = items[i].AddToFromClause(_queryModifier,
_activeUsq);
                    // If current dataSource was added automatically
                    if (_autoAddedDataSources.Contains(currentDataSource))
                        // then move it according to items position
                    {
                        _activeUsq.FromClause.Move(currentDataSource, i);
```

```
                    // delete this dataSource from _autoAddedDataSources
                    _autoAddedDataSources.Remove(currentDataSource);
                }

                // Save last dataSource
                if (i == items.Length - 1)
                    last = currentDataSource;
            }
            Debug.Assert(last != null);

            return new SelectJoinStep(this, last);
        }
        public ISelectFromStep Hint(string hint)
        {
            //TODO: ???
            return this;
        }
        public ISelectStep Select()
        {
            _queryModifier.AddSelectItem(_activeUsq, new SelectItem(_sqlContext,
"*"));
            return this;
        }
        public ISelectStep Select(params string[] strSelectItems)
        {
            if (strSelectItems == null)
                throw new ArgumentNullException("strSelectItems");
            foreach (var strSelectItem in strSelectItems)
            {
                var selectItem = _sqlContext.ParseSelectItem(strSelectItem);
                _queryModifier.AddSelectItem(_activeUsq, new
SelectItem(selectItem, _sqlContext));
            }
            return this;
        }
        public ISelectStep Select(params ISelectItem[] selectItems)
        {
            if (selectItems == null)
                throw new ArgumentNullException("selectItems");
            // Collect dataSources which already exists in query
            var previousDataSourceSet = new HashSet<DataSourceBase>();
            _activeUsq.FromClause.Items.ForEach(ds =>
previousDataSourceSet.Add(ds));
            foreach (var selectItem in selectItems)
            {
                selectItem.AddToSelectList(_queryModifier, _activeUsq);
            }
            // Collect dataSources after adding selectItems
            var currentDataSourceSet = new HashSet<DataSourceBase>();
```

```
                _activeUsq.FromClause.Items.ForEach(ds =>
currentDataSourceSet.Add(ds));
                // and check if it contains new dataSources
                // add them to _autoAddedDataSources

_autoAddedDataSources.Append(currentDataSourceSet.Subtract(previousDataSourceSet));
                return this;
            }
            public ISelectStep SelectDistinct(params ISelectItem[] selectItems)
            {
                if (selectItems == null)
                    throw new ArgumentNullException("selectItems");
                if (_activeUsq.SelectMode == null)
                    _activeUsq.SelectMode = new
SQLSubQuerySelectModeDistinct(_sqlContext);
                foreach (var selectItem in selectItems)
                {
                    selectItem.AddToSelectList(_queryModifier, _activeUsq);
                }
                return this;
            }
            public IField Field(int index)
            {
                if (string.IsNullOrEmpty(_asDerivedTableAlias))
                    _asDerivedTableAlias = CreateUniqueAlias(_queryRoot, "query_");
                return GetField(index);
            }
            public IField Field(string name)
            {
                if (string.IsNullOrEmpty(_asDerivedTableAlias))
                    _asDerivedTableAlias = CreateUniqueAlias(_queryRoot, "query_");
                return GetField(name);
            }
            public IField Field(IField field)
            {
                if (string.IsNullOrEmpty(_asDerivedTableAlias))
                    _asDerivedTableAlias = CreateUniqueAlias(_queryRoot, "query_");
                return GetField(field.FieldName);
            }
            public ITable AsTable(string name)
            {
                return new DerivedTable(this, name, name);
            }
            public IField AsField(string alias)
            {
                return new Field("(" + SQL + ")", null, alias);
            }
            public DataSource AddToFromClause(IQueryModifier queryModifier,
UnionSubQuery unionSubQuery)
```

```
            {
                if (string.IsNullOrEmpty(_asDerivedTableAlias))
                    _asDerivedTableAlias = CreateUniqueAlias(_queryRoot, "query_");
                var derivedTable = (DerivedTable)AsTable(_asDerivedTableAlias);

                return derivedTable.AddToFromClause(queryModifier, unionSubQuery);
            }
            public void AddToSelectList(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
            {
                queryModifier.AddSelectItem(unionSubQuery, new SelectItem(_sqlContext,
"(" + SQL + ")", _asSelectItemAlias));
                _asSelectItemAlias = null;
            }
            private string CreateUniqueAlias(UnionGroup queryRoot, string prefix)
            {
                var random = new Random(GetHashCode());
                var numPrefix = random.Next(0, int.MaxValue);
                var subQueries = queryRoot.GetUnionSubQueryList();
                foreach (var unionSubQuery in subQueries)
                {
                    prefix =
SQLQueryService.Instance.CreateUniqueDataSourceAliasStr(unionSubQuery, prefix +
numPrefix);
                }
                return prefix;
            }
            private Field GetField(string name)
            {
                using (var statistic =
QueryStatisticsService.GetStatistics(_queryRoot))
                {
                    foreach (var outputColumn in statistic.OutputColumns)
                    {
                        if (outputColumn.ColumnName == name)
                            return new QueryField(outputColumn, _asDerivedTableAlias);
                    }
                }
                return null;
            }
            private Field GetField(int index)
            {
                using (var statistic =
QueryStatisticsService.GetStatistics(_queryRoot))
                {
                    return index > statistic.OutputColumns.Count - 1 ?
                        null :
                        new QueryField(statistic.OutputColumns[index],
_asDerivedTableAlias);
```

```
            }
        }
        private List<IField> AllFields()
        {
            var result = new List<IField>();
            using (var statistic =
QueryStatisticsService.GetStatistics(_queryRoot))
            {
                result.AddRange(statistic.OutputColumns.Select(outputColumn => new
QueryField(outputColumn, _asDerivedTableAlias)));
            }
            return result;
        }
        private bool IsCorrectUnionStep(IQueryStatisticsProvider otherSelect)
        {
            var isSameColsCount = OutputColumns.Count ==
otherSelect.OutputColumns.Count;
            var columnsDataTypeSequence = OutputColumns.Select(col =>
col.FieldType);
            var otherSequence = otherSelect.OutputColumns.Select(col =>
col.FieldType);
            var isDataTypesEquals =
columnsDataTypeSequence.SequenceEqual(otherSequence);
            return isSameColsCount && isDataTypesEquals;
        }
        internal void BeforeGetResult()
        {
            foreach (var action in _beforeGetResultConditionsActions)
            {
                action();
            }
        }
        public void Dispose()
        {
            if (_queryStatistics != null)
                _queryStatistics.Dispose();
            _queryRoot.Updated -= _queryRoot_Updated;
            _queryRoot.Dispose();
            _autoAddedDataSources.Clear();
            _beforeGetResultConditionsActions.Clear();
        }
    }
```

Файл AbstractNamespace.cs

```
public abstract class AbstractNamespace : INamespace
    {
        private readonly MetadataList _parentList;
        internal MetadataNamespace Namespace;
        public MetadataType MetadataType { get; protected set; }
        public string Name { get { return GetName(); } }
```

```
            public bool Default { get; set; }
            protected AbstractNamespace(MetadataList parentList, MetadataType
metadataType)
            {
                MetadataType = metadataType;
                _parentList = parentList;
            }
            protected void Initialize()
            {
                var needAddToParent = Namespace == null;
                // Find or create
                Namespace =

_parentList.FindItem<MetadataNamespace>(GetQualifiedName(_parentList.SQLContext)) ??
                    _parentList.FindItem<MetadataNamespace>(GetName()) ??
                    new MetadataNamespace(_parentList, MetadataType)
                    {
                        Name = Name,
                        Default = Default
                    };
                if (needAddToParent)
                    _parentList.Add(Namespace);
            }
            protected T CreateItem<T>()
            {
                var createdType = typeof(T);
                var interfaces = createdType.GetInterfaces();
                if (interfaces.Contains(typeof(ITable)))
                    return Table<T>.Create(Namespace.Items);
                if (interfaces.Contains(typeof(INamespace)))
                    return Create<T>(Namespace.Items);
                return default(T);
            }
            private T Create<T>(MetadataList parentList)
            {
                var ctor = typeof(T).GetConstructor(new[]{typeof(MetadataList)});
                Debug.Assert(ctor != null);
                var ns = ctor.Invoke(new object[] { parentList});
                return (T)ns;
            }
            private string GetName()
            {
                var schemaNameAttribute =
(NameAttribute)GetType().GetCustomAttributes(typeof(NameAttribute),
false).FirstOrDefault();
                if (schemaNameAttribute == null)
                    throw new Exception("Missing NamespaceAttribute");
                return schemaNameAttribute.Name;
            }
```

```
            private string GetQualifiedName(SQLContext ctx)
            {
                using (var parsedName = new SQLQualifiedName(ctx))
                {
                    parsedName.AddName(Name, true);
                    return parsedName.QualifiedName;
                }
            }
        }
```

Файл AbstractRoutine.cs

```
        public abstract class AbstractRoutine : ISubRoutine,  IFromClauseItem,
ISelectItem, IExpression<AbstractRoutine>
        {
            private readonly Dictionary<IParameter, object> _params = new
Dictionary<IParameter, object>();
            private readonly MetadataObject MetadataObject;
            protected AbstractRoutine(AbstractNamespace @namespace)
            {
                var needAddToParent = MetadataObject == null;
                Schema = @namespace;
                //Find or create
                MetadataObject =

@namespace.Namespace.FindItem<MetadataObject>(GetQualifiedName(@namespace.Namespace.S
QLContext)) ??
                    @namespace.Namespace.FindItem<MetadataObject>(GetName()) ??
                    new MetadataObject(@namespace.Namespace.Items,
MetadataType.Procedure)
                    {
                        Name = Name
                    };
                if (needAddToParent)
                    @namespace.Namespace.Items.Add(MetadataObject);
            }
            public INamespace Schema { get; private set; }
            public string Name
            {
                get { return GetName(); }
            }
            public string Alias { get; set; }
            public object Expression
            {
                get { return BuildExpression(); }
            }
            public void SetParameter(string name, object value)
            {
                var parameter = Parameters.FirstOrDefault(p => p.Name == name);
                if (parameter == null)
```

```
                throw new Exception("Parameter not found");
            _params[parameter] = value;
        }
        public List<IParameter> Parameters
        {
            get { return _params.Keys.ToList(); }
        }

        public DataSource AddToFromClause(IQueryModifier queryModifier,
UnionSubQuery unionSubQuery)
        {
            return queryModifier.AddDataSource(unionSubQuery, BuildExpression(),
Alias);
        }
        public void AddToSelectList(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
        {
            queryModifier.AddSelectItem(unionSubQuery, new
SelectItem(unionSubQuery.SQLContext, BuildExpression(), Alias));
        }
        public AbstractRoutine As(string alias)
        {
            var ctor = GetType().GetConstructor(new
[]{typeof(AbstractNamespace)});
            Debug.Assert(ctor != null, "Constructor not found");
            var clonedRoutine = (AbstractRoutine)ctor.Invoke(new
object[]{Schema});
            clonedRoutine.Alias = alias;
            return clonedRoutine;
        }
        private string BuildExpression()
        {
            var strParams = new StringBuilder(Name + "(");
            foreach (var parameter in _params)
            {
                strParams.Append(parameter.Value);
                strParams.Append(", ");
            }
            // If this routine has params
            // remove last space with comma
            if (Parameters.Count != 0)
                strParams.Length -= 2;
            strParams.Append(")");
            return strParams.ToString();
        }
        private string GetName()
        {
```

```
                var nameAttribute =
(NameAttribute)GetType().GetCustomAttributes(typeof(NameAttribute),
false).FirstOrDefault();
                if (nameAttribute == null)
                    throw new Exception("Missing NameAttribute in class " +
GetType().Name);
                return nameAttribute.Name;
            }
            protected void RegisterParams()
            {
                var parameters = GetType()
                    .GetProperties(BindingFlags.DeclaredOnly | BindingFlags.Public |
BindingFlags.Instance)
                    .Where(propertyInfo => propertyInfo.PropertyType ==
typeof(RoutineParam))
                    .Select(propertyInfo => (RoutineParam)propertyInfo.GetValue(this,
null));
                foreach (var parameter in parameters)
                {
                    _params.Add(parameter, null);
                }
            }
            private string GetQualifiedName(SQLContext ctx)
            {
                using (var parsedName = new SQLQualifiedName(ctx))
                {
                    parsedName.AddName(Name, true);
                    return parsedName.QualifiedName;
                }
            }
        }
```

Файл Table.cs

```
        /// <inheritdoc cref="ITable{T}"/>
        public class Table<T> : Table, ITable<T>
        {
            private List<IField> _fields = new List<IField>();
            private readonly MetadataList _parentMetadataList;
            private string _alias = string.Empty;
            private bool _isFieldsInitialized;

            public MetadataObject MetadataObject { get; private set; }
            public override string TableName { get { return MetadataObject.Name; } }
            public override string Schema { get { return MetadataObject.Schema.Name; }
}
            public override string Alias { get { return _alias; } set { _alias =
value; } }
            public override List<IField> Fields {
                get
```

```
            {
                if (_isFieldsInitialized) return _fields;

                _fields = CollectFieldsFromInheritor();
                _isFieldsInitialized = true;
                return _fields;
            }
        }
        protected Table(MetadataList metadataList) : base("", "")
        {
            var needAddToParent = MetadataObject == null;
            _parentMetadataList = metadataList;

            // Find or create
            MetadataObject =

metadataList.FindItem<MetadataObject>(GetQualifiedName(metadataList.SQLContext)) ??
                metadataList.FindItem<MetadataObject>(GetNameFromAttribute()) ??
                new MetadataObject(metadataList, MetadataType.Table)
                {
                    Name = GetNameFromAttribute(),
                };
            if (needAddToParent)
                metadataList.Tables.Add(MetadataObject);
        }
        public string GetQualifiedName(SQLContext sqlContext)
        {
            var name = GetNameFromAttribute();
            using (var parsedName = new SQLQualifiedName(sqlContext))
            {
                parsedName.AddName(name, true);
                return parsedName.QualifiedName;
            }
        }
        private string GetNameFromAttribute()
        {
            var tableNameAttribute =
(NameAttribute)typeof(T).GetCustomAttributes(typeof(NameAttribute), false).First();
            if (tableNameAttribute == null)
                throw new Exception("Missing NameAttribute in class " +
typeof(T).Name);
            return tableNameAttribute.Name;
        }
        public static T Create(MetadataList metadataList)
        {
            var ctor = typeof(T).GetConstructor(new[] { typeof(MetadataList) });
            Debug.Assert(ctor != null);
            return (T)ctor.Invoke(new object[] { metadataList });
        }
```

```
            public new T As(string alias)
            {
                var aliasedTable = (ITable)Create(_parentMetadataList);
                aliasedTable.Alias = alias;
                return (T)aliasedTable;
            }
            public override DataSource AddToFromClause(IQueryModifier queryModifier,
UnionSubQuery unionSubQuery)
            {
                var dataSources =
unionSubQuery.FindDataSources(MetadataObject).Where(ds => ds.Alias ==
Alias).ToList();

                return dataSources.Any() ?
                       dataSources.First() :
                       queryModifier.AddDataSource(unionSubQuery, MetadataObject,
Alias);
            }
            public override IField Field(string name)
            {
                return Fields.FirstOrDefault(f => f.FieldName == name);
            }
            public override IField Field(IField field)
            {
                return Fields.FirstOrDefault(f => f.FieldName == field.FieldName &&
f.NameInQuery == field.NameInQuery);
            }
            private List<IField> CollectFieldsFromInheritor()
            {
                return GetType()
                    .GetProperties(BindingFlags.Public | BindingFlags.DeclaredOnly |
BindingFlags.Instance)
                    .Where(propInfo => propInfo.PropertyType.BaseType ==
typeof(Field))
                    .Select(propInfo => (IField) propInfo.GetValue(this, null))
                    .ToList();
            }
        }
        public class Table : ITable
        {
            private readonly List<IField> _fields = new List<IField>();
            private string _alias;
            public virtual List<IField> Fields {
                get { return _fields; }
            }
            public virtual string TableName { get; private set; }
            public virtual string Schema { get; private set; }
            public virtual string Alias
            {
```

```
            get { return _alias;}
            set { _alias = value; }
        }
        public Table(string name, string schema ,string alias = null)
        {
            TableName = name;
            Schema = schema;
            _alias = alias ?? string.Empty;
        }
        public virtual string SelectAllFields()
        {
            if (!string.IsNullOrEmpty(Alias))
                return Alias + ".*";
            return TableName + ".*";
        }
        public virtual DataSource AddToFromClause(IQueryModifier queryModifier,
UnionSubQuery unionSubQuery)
        {
            return queryModifier.AddDataSource(unionSubQuery, TableName, Alias);
        }
        public virtual IField Field(int index)
        {
            return Fields[index];
        }
        public virtual IField Field(string name)
        {
            var  field = Fields.FirstOrDefault(f => f.FieldName == name);
            if (field != null) return field;
            field = new Field(name, TableName, Alias);
            _fields.Add(field);
            return field;
        }
        public virtual IField Field(IField fieldToFind)
        {
            var field = Fields.FirstOrDefault(f => f.FieldName ==
fieldToFind.FieldName && f.NameInQuery == fieldToFind.NameInQuery);
            if (field != null) return field;
            field = new Field(fieldToFind.FieldName, fieldToFind.TableAlias,
fieldToFind.Alias);
            _fields.Add(field);
            return field;
        }
        public virtual ITable AsTable(string name)
        {
            return As(name);
        }
        public virtual ITable As(string alias)
        {
            return new Table(TableName, Schema, alias);
```

```
            }
            public override string ToString()
            {
                return TableName;
            }
        }
```

Файл Field.cs

```
    /// <inheritdoc cref="IField"/>
        public class Field : ConditionFactory,  IField
        {
            private bool _needToQuote;

            protected internal Field(string fieldName, string tableAlias, string alias
= null)
            {
                TableAlias = tableAlias;
                FieldName = fieldName;
                Alias = alias;
                _needToQuote = FieldName.Contains(" ");
            }
            /// <inheritdoc cref="IField.FieldName"/>
            public string FieldName { get; private set; }
            /// <inheritdoc cref="IField.TableAlias"/>
            public virtual string TableAlias { get; private set; }
            /// <inheritdoc cref="IField.NameInQuery"/>
            public virtual string NameInQuery
            {
                get
                {
                    if (!string.IsNullOrEmpty(TableAlias))
                        return TableAlias + "." + FieldName;
                    return FieldName;
                }
            }
            /// <inheritdoc cref="IField.As"/>
            public virtual IField As(string alias)
            {
                return new Field(FieldName, TableAlias, alias);
            }
            /// <inheritdoc cref="IExpression.Expression"/>
            public override object Expression
            {
                get { return NameInQuery; }
            }
            /// <inheritdoc cref="ISelectItem.AddToSelectList"/>
            public virtual void AddToSelectList(IQueryModifier queryModifier,
UnionSubQuery unionSubQuery)
            {
```

```
                var sqlContext = unionSubQuery.SQLContext;
                if (_needToQuote && GetType() == typeof(Field))
                {
                    FieldName = Helpers.CreateQuotedIdentifier(FieldName,
sqlContext.SyntaxProvider.QuoteBegin,
                        sqlContext.SyntaxProvider.QuoteEnd);
                    _needToQuote = false;
                }
                var selectItem = sqlContext.ParseSelectItem(NameInQuery);
                queryModifier.AddSelectItem(unionSubQuery, new SelectItem(selectItem,
sqlContext){Alias = Alias});
            }
            /// <inheritdoc cref="IGroupingItem.AddToGroupByList"/>
            public virtual void AddToGroupByList(IQueryModifier query, UnionSubQuery
unionSubQuery)
            {
                var sqlExpressionItem =
unionSubQuery.SQLContext.ParseExpression(NameInQuery);
                unionSubQuery.GroupingList.Add(new GroupingItem(sqlExpressionItem,
unionSubQuery.FromColumns, unionSubQuery.SQLContext));
            }
            /// <inheritdoc cref="IOrderByItem.AddToOrderByList"/>
            public virtual void AddToOrderByList(IQueryModifier queryModifier,
UnionSubQuery unionSubQuery)
            {
                var sqlOrderByItem =
unionSubQuery.SQLContext.ParseOrderByItem(NameInQuery);
                queryModifier.AddOrderByItem(unionSubQuery, new
OrderByItem(sqlOrderByItem, unionSubQuery.SQLContext));
            }
            public IAggregatedItem Sum()
            {
                return new AggregateItem(AggregateFunction.Sum, NameInQuery);
            }
            public IAggregatedItem SumDistinct()
            {
                return new AggregateItem(AggregateFunction.Sum, "distinct " +
NameInQuery);
            }
            public IAggregatedItem Max()
            {
                return new AggregateItem(AggregateFunction.Max, NameInQuery);
            }
            public IAggregatedItem MaxDistinct()
            {
                return new AggregateItem(AggregateFunction.Max, "distinct " +
NameInQuery);
            }
            public IAggregatedItem Min()
```

```
            {
                return new AggregateItem(AggregateFunction.Min, NameInQuery);
            }
            public IAggregatedItem MinDistinct()
            {
                return new AggregateItem(AggregateFunction.Min, "distinct " +
NameInQuery);
            }
            public IAggregatedItem Avg()
            {
                return new AggregateItem(AggregateFunction.Avg, NameInQuery);
            }
            public IAggregatedItem AvgDistinct()
            {
                return new AggregateItem(AggregateFunction.Avg, "distinct " +
NameInQuery);
            }
            public IAggregatedItem Count()
            {
                return new AggregateItem(AggregateFunction.Count, NameInQuery);
            }
            public IAggregatedItem CountDistinct()
            {
                return new AggregateItem(AggregateFunction.Count, "distinct " +
NameInQuery);
            }
            public ISortingItem Asc()
            {
                return new SortingItem(SortingOperator.Asc, this);
            }
            public ISortingItem Desc()
            {
                return new SortingItem(SortingOperator.Desc, this);
            }
        }
```

Файл TableField.cs

```
    public class TableField<T> : Field
        {
            public static TableField<T> NULL
            {
                get { return null; }
            }
            internal readonly MetadataField MetadataField;
            public bool ReadOnly
            {
                get { return MetadataField.ReadOnly; }
                set { MetadataField.ReadOnly = value; }
            }
```

```
            public bool PrimaryKey
            {
                get { return MetadataField.PrimaryKey; }
                set { MetadataField.PrimaryKey = value; }
            }
            public bool Nullable
            {
                get { return MetadataField.Nullable; }
                set { MetadataField.Nullable = value; }
            }
            public string FieldTypeName
            {
                get { return MetadataField.FieldTypeName; }
                set { MetadataField.FieldTypeName = value; }
            }
            public string AltName
            {
                get { return MetadataField.AltName; }
                set { MetadataField.AltName = value; }
            }
            public long Size
            {
                get { return MetadataField.Size; }
                set { MetadataField.Size = value; }
            }
            public int Precision
            {
                get { return MetadataField.Precision; }
                set { MetadataField.Precision = value; }
            }
            public int Scale
            {
                get { return MetadataField.Scale; }
                set { MetadataField.Scale = value; }
            }
            public override string NameInQuery
            {
                get
                {
                    if (!string.IsNullOrEmpty(TableAlias))
                        return TableAlias + "." + FieldName;

                    return Table.TableName + "." + FieldName;
                }
            }
            protected override Condition GetCondition(string value)
            {
                var logicalExpression =
MetadataField.SQLContext.ParseLogicalExpression(NameInQuery + value);
```

```
                return new
Condition(logicalExpression.GetSQL(MetadataField.SQLContext.SQLGenerationOptionsForSe
rver));
            }
            public ITable Table { get; private set; }
            private TableField(ITable table, MetadataField metadataField) :
base(metadataField.Name, table.Alias)
            {
                Table = table;
                MetadataField = metadataField;
            }
            public TableField(string fieldName, ITable table, MetadataObject
metadataObject) : base(fieldName, table.Alias)
            {
                Table = table;
                var needAddToParent = MetadataField == null;
                MetadataField =
                    metadataObject.Items.FindItem<MetadataField>(fieldName) ??
                    new MetadataField(metadataObject.Items)
                    {
                        Name = fieldName,
                        FieldType = Helpers.TypeToDbType(typeof(T))
                    };
                if (needAddToParent)
                    metadataObject.Items.Fields.Add(MetadataField);
            }
            public override string TableAlias
            {
                get { return Table.Alias; }
            }

            public override IField As(string alias)
            {
                return new TableField<T>(Table, MetadataField){Alias = alias};
            }
            public static TableField<T> Create(string fieldName, ITable table,
MetadataObject metadataObject)
            {
                var metadataField =
                    metadataObject.Items.FindItem<MetadataField>(fieldName) ??
                    new MetadataField(metadataObject.Items)
                    {
                        Name = fieldName,
                        FieldType = Helpers.TypeToDbType(typeof(T))
                    };
                metadataObject.Items.Fields.Add(metadataField);
                return new TableField<T>(table, metadataField);
            }
            public static Condition operator ==(TableField<T> lhs, TableField<T> rhs)
```

```
{
    if (!ReferenceEquals(lhs, null) && ReferenceEquals(rhs, null))
    {
        return new Condition(lhs.NameInQuery + " Is Null");
    }
    if (ReferenceEquals(lhs, null) && !ReferenceEquals(rhs, null))
    {
        return new Condition(rhs.NameInQuery + " Is Null");
    }
    if (ReferenceEquals(lhs, null) && ReferenceEquals(rhs, null))
        throw new ArgumentNullException();
    return lhs.Equal(rhs);
}

public static Condition operator !=(TableField<T> lhs, TableField<T> rhs)
{
    if (!ReferenceEquals(lhs, null) && ReferenceEquals(rhs, null))
    {
        return new Condition(lhs.NameInQuery + " Is Not Null");
    }
    if (ReferenceEquals(lhs, null) && !ReferenceEquals(rhs, null))
    {
        return new Condition(rhs.NameInQuery + " Is Not Null");
    }
    if (ReferenceEquals(lhs, null) && ReferenceEquals(rhs, null))
        throw new ArgumentNullException();
    return lhs.Not_Equal(rhs);
}
public static Condition operator >(TableField<T> lhs, TableField<T> rhs)
{
    if (ReferenceEquals(lhs, null) || ReferenceEquals(rhs, null))
        throw new ArgumentNullException();
    return lhs.Greater(rhs);
}
public static Condition operator <(TableField<T> lhs, TableField<T> rhs)
{
    if (ReferenceEquals(lhs, null) || ReferenceEquals(rhs, null))
        throw new ArgumentNullException();
    return lhs.Less(rhs);
}
public static Condition operator >=(TableField<T> lhs, TableField<T> rhs)
{
    if (ReferenceEquals(lhs, null) || ReferenceEquals(rhs, null))
        throw new ArgumentNullException();
    return lhs.GreaterEqual(rhs);
}
public static Condition operator <=(TableField<T> lhs, TableField<T> rhs)
{
    if (ReferenceEquals(lhs, null) || ReferenceEquals(rhs, null))
```

```
            throw new ArgumentNullException();
        return lhs.LessEqual(rhs);
    }
    public static Condition operator ==(TableField<T> lhs, int rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Equal(rhs);
    }
    public static Condition operator !=(TableField<T> lhs, int rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();

        return lhs.Not_Equal(rhs);
    }
    public static Condition operator >(TableField<T> lhs, int rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Greater(rhs);
    }
    public static Condition operator <(TableField<T> lhs, int rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Less(rhs);
    }
    public static Condition operator >=(TableField<T> lhs, int rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.GreaterEqual(rhs);
    }
    public static Condition operator <=(TableField<T> lhs, int rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.LessEqual(rhs);
    }
    public static Condition operator ==(TableField<T> lhs, double rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Equal(rhs);
    }
    public static Condition operator !=(TableField<T> lhs, double rhs)
    {
        if (ReferenceEquals(lhs, null))
```

```
            throw new ArgumentNullException();
        return lhs.Not_Equal(rhs);
    }
    public static Condition operator >(TableField<T> lhs, double rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Greater(rhs);
    }
    public static Condition operator <(TableField<T> lhs, double rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();

        return lhs.Less(rhs);
    }
    public static Condition operator >=(TableField<T> lhs, double rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.GreaterEqual(rhs);
    }
    public static Condition operator <=(TableField<T> lhs, double rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.LessEqual(rhs);
    }
    public static Condition operator ==(TableField<T> lhs, string rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Equal(rhs);
    }
    public static Condition operator !=(TableField<T> lhs, string rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Not_Equal(rhs);
    }
    public static Condition operator >(TableField<T> lhs, string rhs)
    {
        if (ReferenceEquals(lhs, null))
            throw new ArgumentNullException();
        return lhs.Greater(rhs);
    }
    public static Condition operator <(TableField<T> lhs, string rhs)
    {
        if (ReferenceEquals(lhs, null))
```

```
                throw new ArgumentNullException();
            return lhs.Less(rhs);
        }
        public static Condition operator >=(TableField<T> lhs, string rhs)
        {
            if (ReferenceEquals(lhs, null))
                throw new ArgumentNullException();
            return lhs.GreaterEqual(rhs);
        }
        public static Condition operator <=(TableField<T> lhs, string rhs)
        {
            if (ReferenceEquals(lhs, null))
                throw new ArgumentNullException();

            return lhs.LessEqual(rhs);
        }
        public override void AddToSelectList(IQueryModifier queryModifier,
UnionSubQuery unionSubQuery)
        {
            // Try to find the data source referenced by this field,
            // if there is no one then add it.
            var dataSource =
unionSubQuery.FindDataSources(MetadataField.Object).FirstOrDefault(ds => ds.Alias ==
Table.Alias) ??
                                queryModifier.AddDataSource(unionSubQuery,
MetadataField.Object, Table.Alias);
            var fromColumn = unionSubQuery.FromColumns.Find(dataSource,
FieldName);
            Debug.Assert(fromColumn != null);
            var selectItem = new SelectItem(MetadataField.SQLContext, fromColumn,
Alias);
            queryModifier.AddSelectItem(unionSubQuery, selectItem);
        }
        public override string ToString()
        {
            return NameInQuery;
        }
        private bool Equals(TableField<T> other)
        {
            return Equals(MetadataField, other.MetadataField) &&
                   Equals(Table, other.Table) &&
                   string.Equals(Alias, other.Alias);
        }
        public override bool Equals(object obj)
        {
            if (ReferenceEquals(null, obj)) return false;
            if (ReferenceEquals(this, obj)) return true;
            if (obj.GetType() != this.GetType()) return false;
            return Equals((TableField<T>)obj);
```

```
            }
            public override int GetHashCode()
            {
                unchecked
                {
                    var hashCode = (MetadataField != null ?
MetadataField.GetHashCode() : 0);
                    hashCode = (hashCode * 397) ^ (Table != null ? Table.GetHashCode()
: 0);
                    hashCode = (hashCode * 397) ^ (Alias != null ? Alias.GetHashCode()
: 0);
                    return hashCode;
                }
            }
        }
    Файл SelectAfterLimitOffsetStep.cs
    public class SelectAfterLimitOffsetStep : ISelectAfterLimitOffsetStep
    {
        private readonly IDSLContext _dslContext;
        public StatisticsDatabaseObjectList UsedDatabaseObjects
        {
            get { return _dslContext.UsedDatabaseObjects; }
        }
        public StatisticsFieldList UsedDatabaseObjectFields
        {
            get { return _dslContext.UsedDatabaseObjectFields; }
        }
        public StatisticsOutputColumnList OutputColumns
        {
            get { return _dslContext.OutputColumns; }
        }
        public List<IField> Fields { get { return _dslContext.Fields; } }
        public QueryRoot Result { get { return _dslContext.Result; }}
        public string SQL { get { return _dslContext.SQL; } }
        public ISelectResult As(string alias)
        {
            return _dslContext.As(alias);
        }
        public SelectAfterLimitOffsetStep(IDSLContext dslContext)
        {
            _dslContext = dslContext;
        }
        public ISelectOrderByStep Union(Action<ISelectStep> selectAction)
        {
            return _dslContext.Union(selectAction);
        }
        public ISelectOrderByStep UnionAll(Action<ISelectStep> selectAction)
        {
            return _dslContext.UnionAll(selectAction);
```

```
        }
        public ISelectOrderByStep Union(ISelectResult @select)
        {
            return _dslContext.Union(select);
        }
        public ISelectOrderByStep UnionAll(ISelectResult @select)
        {
            return _dslContext.UnionAll(select);
        }
        public ISelectUnionStep Limit(int val)
        {
            return _dslContext.Limit(val);
        }
        public ISelectUnionStep Limit(string limit)
        {
            return _dslContext.Limit(limit);
        }
        public IField Field(int index)
        {
            return _dslContext.Field(index);
        }
        public IField Field(string name)
        {
            return _dslContext.Field(name);
        }
        public IField Field(IField field)
        {
            return _dslContext.Field(field);
        }
        public ITable AsTable(string name)
        {
            return _dslContext.AsTable(name);
        }
        public IField AsField(string alias)
        {
            return _dslContext.AsField(alias);
        }
        public DataSource AddToFromClause(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
        {
            return _dslContext.AddToFromClause(queryModifier, unionSubQuery);
        }
        public void AddToSelectList(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
        {
            _dslContext.AddToSelectList(queryModifier, unionSubQuery);
        }
    }
```

Файл SelectConditionStep.cs

```
class SelectConditionStep : ISelectConditionStep
{
    private readonly ISelectStep _dslContext;
    // Reference to Where or Having conditionsList in UnionSubQuery.
    private readonly ConditionsList _parentConditionsList;
    // Base list in which we add all the conditions.
    private ConditionsList _currentList;
    private bool _wasAddedToParent;
    public StatisticsDatabaseObjectList UsedDatabaseObjects
    {
        get { return _dslContext.UsedDatabaseObjects; }
    }

    public StatisticsFieldList UsedDatabaseObjectFields
    {
        get { return _dslContext.UsedDatabaseObjectFields; }
    }
    public StatisticsOutputColumnList OutputColumns
    {
        get { return _dslContext.OutputColumns; }
    }
    public List<IField> Fields { get { return _dslContext.Fields; } }
    public QueryRoot Result { get { return _dslContext.Result; } }
    public string SQL
    {
        get { return _dslContext.SQL; }
    }
    public ISelectResult As(string alias)
    {
        return _dslContext.As(alias);
    }
    public SelectConditionStep(IDSLContext dslContext, ConditionsList
parentConditionsList, ConditionsList current)
    {
        _dslContext = dslContext;
        _parentConditionsList = parentConditionsList;
        _currentList = current;
    }
    public ISelectOrderByStep Union(Action<ISelectStep> selectAction)
    {
        return _dslContext.Union(selectAction);
    }
    public ISelectOrderByStep UnionAll(Action<ISelectStep> selectAction)
    {
        return _dslContext.UnionAll(selectAction);
    }
    public ISelectOrderByStep Union(ISelectResult select)
```

```
{
    return _dslContext.Union(select);
}
public ISelectOrderByStep UnionAll(ISelectResult select)
{
    return _dslContext.UnionAll(select);
}

public ISelectOffsetStep Limit(int val)
{
    return _dslContext.Limit(val);
}
public ISelectOffsetStep Limit(string limit)
{
    return _dslContext.Limit(limit);
}
public ISelectAfterLimitOffsetStep Offset(int val)
{
    return _dslContext.Offset(val);
}
public ISelectAfterLimitOffsetStep Offset(string offset)
{
    return _dslContext.Offset(offset);
}
public ISelectHavingConditionStep Having(ConditionBase condition)
{
    return _dslContext.Having(condition);
}
public ISelectHavingConditionStep Having(Condition condition)
{
    return _dslContext.Having(condition);
}
public ISelectHavingConditionStep Having(string condition)
{
    return _dslContext.Having(condition);
}
public ISelectHavingStep GroupBy(params string[] aggregatedItems)
{
    return _dslContext.GroupBy(aggregatedItems);
}
public ISelectHavingStep GroupBy(params IGroupingItem[] groupingItems)
{
    return _dslContext.GroupBy(groupingItems);
}
public ISelectLimitStep OrderBy(params string[] sortingItems)
{
    return _dslContext.OrderBy(sortingItems);
}
```

```
        public ISelectLimitStep OrderBy(params IOrderByItem[] fields)
        {
            return _dslContext.OrderBy(fields);
        }
        public ISelectConditionStep Or(string condition)
        {
            var logicalExpression =
_parentConditionsList.SqlContext.ParseLogicalExpression(condition);
            var conditionItem = new ConditionItem(logicalExpression,
_parentConditionsList.SqlContext);

            return Or(conditionItem);
        }
        public ISelectConditionStep Or(Condition condition)
        {
            return Or(condition.ExpressionStr);
        }
        public ISelectConditionStep Or(ConditionItem condition)
        {
            return AddCondition(condition, JunctionOperator.OR);
        }
        public ISelectConditionStep OrNot(string condition)
        {
            return Or("Not " + condition);
        }
        public ISelectConditionStep OrNot(Condition condition)
        {
            return OrNot(condition.ExpressionStr);
        }
        public ISelectConditionStep OrNot(ConditionItem condition)
        {
            condition.Expression = "Not " + condition.Expression;

            return Or(condition);
        }
        public ISelectConditionStep OrExists(string expression)
        {
            var logicalExpression =
_parentConditionsList.SqlContext.ParseLogicalExpression("Exists(" + expression +
")");
            var conditionItem = new ConditionItem(logicalExpression,
_parentConditionsList.SqlContext);

            return Or(conditionItem);
        }
        public ISelectConditionStep OrExists(Action<ISelectStep> selectAction)
        {
            using (var dsl = new DSLContext(_dslContext.Result.SQLContext))
            {
```

```
                selectAction(dsl);

                return OrExists(dsl.SQL);
            }
        }
        public ISelectConditionStep OrExists(ISelectResult select)
        {
            return OrExists(select.SQL);
        }
        public ISelectConditionStep OrNotExists(string expression)
        {
            var logicalExpression =
_parentConditionsList.SqlContext.ParseLogicalExpression("Not Exists(" + expression +
")");
            var conditionItem = new ConditionItem(logicalExpression,
_parentConditionsList.SqlContext);

            return Or(conditionItem);
        }
        public ISelectConditionStep OrNotExists(Action<ISelectStep> selectAction)
        {
            using (var dslContext = new DSLContext(Result.SQLContext))
            {
                selectAction(dslContext);

                return OrNotExists(dslContext.SQL);
            }
        }
        public ISelectConditionStep OrNotExists(ISelectResult select)
        {
            return OrNotExists(select.SQL);
        }
        public ISelectConditionStep And(string condition)
        {
            var logicalExpression =
_parentConditionsList.SqlContext.ParseLogicalExpression(condition);
            var conditionItem = new ConditionItem(logicalExpression,
_parentConditionsList.SqlContext);

            return And(conditionItem);
        }
        public ISelectConditionStep And(Condition condition)
        {
            return And(condition.ExpressionStr);
        }
        public ISelectConditionStep And(ConditionItem condition)
        {
            return AddCondition(condition, JunctionOperator.AND);
        }
```

```
public ISelectConditionStep AndNot(string condition)
{
    return And("Not " + condition);
}
public ISelectConditionStep AndNot(Condition condition)
{
    return AndNot(condition.ExpressionStr);
}
public ISelectConditionStep AndNot(ConditionItem condition)
{
    condition.Expression = "Not " + condition.Expression;

    return And(condition);
}
public ISelectConditionStep AndExists(string expression)
{
    return And("Exists(" + expression + ")");
}
public ISelectConditionStep AndExists(Action<ISelectStep> selectAction)
{
    var newSelectStep = new DSLContext(Result.SQLContext);
    {
        selectAction(newSelectStep);
    }
    return AndExists(newSelectStep.SQL);
}

public ISelectConditionStep AndExists(ISelectResult select)
{
    return AndExists(select.SQL);
}
public ISelectConditionStep AndNotExists(string expression)
{
    return And("Not Exists(" + expression + ")");
}
public ISelectConditionStep AndNotExists(Action<ISelectStep> selectAction)
{
    using (var dsl = new DSLContext(Result.SQLContext))
    {
        selectAction(dsl);

        return AndNotExists(dsl.SQL);
    }
}
public ISelectConditionStep AndNotExists(ISelectResult select)
{
    return AndNotExists(select.SQL);
}
internal void AddConditionsToParent()
```

```
        {
            if (!_wasAddedToParent)
            {
_dslContext.QueryModifier.AddConditionItem(_dslContext.ActiveUnionSubQuery,
_parentConditionsList, _currentList);
                _wasAddedToParent = true;
                _currentList = null;
            }
        }
        public IField Field(int index)
        {
            return _dslContext.Field(index);
        }
        public IField Field(string name)
        {
            return _dslContext.Field(name);
        }
        public IField Field(IField field)
        {
            return _dslContext.Field(field);
        }
        public ITable AsTable(string name)
        {
            return _dslContext.AsTable(name);
        }
        public IField AsField(string alias)
        {
            return _dslContext.AsField(alias);
        }
        public DataSource AddToFromClause(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
        {
            return _dslContext.AddToFromClause(queryModifier, unionSubQuery);
        }
        public void AddToSelectList(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
        {
            _dslContext.AddToSelectList(queryModifier, unionSubQuery);
        }
        // Base method to add condition in current list.
        // condition - condition that we trying to add.
        // junctionOperator - operator that we trying to bind the condition.
        private ISelectConditionStep AddCondition(ConditionItem condition,
JunctionOperator junctionOperator)
        {
            // if currentList is null, remove first list from the parent
            // and make it current.
            if (_currentList == null)
```

```
            {
                _wasAddedToParent = false;
                var firstList = (ConditionsList)_parentConditionsList[0];
                _currentList = firstList.Clone();
                _parentConditionsList.Remove(firstList);
            }
            // if operators are same we can add condition
            // to current list.
            if (_currentList.Operator == junctionOperator)
            {
                _currentList.Add(condition);
                return this;
            }
            // Else, we must create new list with given junction operator then
            // add in new list given condition and _currentList then
            // make newList current.
            var newList = new
ConditionsList(_dslContext.ActiveUnionSubQuery.SQLContext) { Operator =
junctionOperator };
            newList.Add(_currentList);
            newList.Add(condition);
            _currentList = newList;
            return this;
        }
    }
```

Файл SelectJoinOnStep.cs

```
    public class SelectJoinOnStep : ISelectJoinOnStep
    {
        private readonly IDSLContext _dslContext;
        private readonly DataSource _left;
        private readonly DataSource _right;
        private readonly JoinType _joinType;
        public SelectJoinOnStep(IDSLContext dslContext, DataSource left, DataSource
right, JoinType joinType)
        {
            _dslContext = dslContext;
            _left = left;
            _right = right;
            _joinType = joinType;
        }
        public ISelectOnConditionStep On(string condition)
        {
            var link = _dslContext.QueryModifier.AddLink(_left, _right, condition,
LinkPlace.From);
            switch (_joinType)
            {
                case JoinType.Join:
```

```
                link.LeftType = LinkSideType.Inner;
                link.RightType = LinkSideType.Inner;
                break;
            case JoinType.CrossJoin:
                link.LeftType = LinkSideType.Inner;
                link.RightType = LinkSideType.Inner;
                break;
            case JoinType.FullOuterJoin:
                link.LeftType = LinkSideType.Outer;
                link.RightType = LinkSideType.Outer;
                break;
            case JoinType.LeftOuterJoin:
                link.LeftType = LinkSideType.Outer;
                link.RightType = LinkSideType.Inner;
                break;
            case JoinType.RightOuterJoin:
                link.LeftType = LinkSideType.Inner;
                link.RightType = LinkSideType.Outer;
                break;
        }
        return new SelectOnConditionStep(_dslContext, link);
    }
    public ISelectOnConditionStep On(Condition condition)
    {
        return On(condition.ExpressionStr);
    }
    public ISelectJoinStep OnKey()
    {
        var leftDataSourcePK = _left
            .MetadataObject
            .FindItems<MetadataField>(MetadataType.Field)
            .FirstOrDefault(field => field.PrimaryKey);
        var rightDataSourcePK = _right.MetadataObject
            .FindItems<MetadataField>(MetadataType.Field)
            .FirstOrDefault(field => field.PrimaryKey);
        // Try to find foreign key associated with left dataSource primary key
        var refField = string.Empty;
        if (leftDataSourcePK != null)
        {
            if (IsPrimaryKeyReferenced(_right, _left, leftDataSourcePK.Name))
                refField = leftDataSourcePK.Name;
        }
        // if not found, try find with right dataSource
        if (rightDataSourcePK != null && string.IsNullOrEmpty(refField))
        {
            if (IsPrimaryKeyReferenced(_left, _right, rightDataSourcePK.Name))
                refField = rightDataSourcePK.Name;
        }
        // Add link if primary key referenced
```

```
            if (!string.IsNullOrEmpty(refField))
                _dslContext.QueryModifier.AddLink(_left, refField, _right, refField,
LinkPlace.From);

            return new SelectJoinStep(_dslContext, _right);
        }
        private bool IsPrimaryKeyReferenced(DataSource left, DataSource right, string
primaryKey)
        {
            var leftDataSourceForeignKeys =

left.MetadataObject.FindItems<MetadataForeignKey>(MetadataType.ForeignKey);
            var foreignKey = leftDataSourceForeignKeys.Find(fk => fk.ReferencedObject
== right.MetadataObject &&

fk.ReferencedFields.Find(primaryKey));
            return foreignKey != null;
        }
    }
```

Файл SelectJoinStep.cs

```
    public class SelectJoinStep : ISelectJoinStep
    {
        private readonly DataSource _dataSource;
        private readonly IDSLContext _dslContext;
        public StatisticsDatabaseObjectList UsedDatabaseObjects
        {
            get { return _dslContext.UsedDatabaseObjects; }
        }
        public StatisticsFieldList UsedDatabaseObjectFields
        {
            get { return _dslContext.UsedDatabaseObjectFields; }
        }
        public StatisticsOutputColumnList OutputColumns
        {
            get { return _dslContext.OutputColumns; }
        }
        public QueryRoot Result { get { return _dslContext.Result; } }
        public string SQL { get { return _dslContext.SQL; } }
        public List<IField> Fields { get { return _dslContext.Fields; } }
        public SelectJoinStep(IDSLContext dslContext, DataSource dataSource)
        {
            _dslContext = dslContext;
            _dataSource = dataSource;
        }
        public ISelectResult As(string alias)
        {
            return _dslContext.As(alias);
```

```
}
public ISelectOrderByStep Union(Action<ISelectStep> selectAction)
{
    return _dslContext.Union(selectAction);
}
public ISelectOrderByStep UnionAll(Action<ISelectStep> selectAction)
{
    return _dslContext.UnionAll(selectAction);
}
public ISelectOrderByStep Union(ISelectResult select)
{
    return _dslContext.Union(select);
}

public ISelectOrderByStep UnionAll(ISelectResult select)
{
    return _dslContext.UnionAll(select);
}
public ISelectOffsetStep Limit(int val)
{
    return _dslContext.Limit(val);
}
public ISelectOffsetStep Limit(string limit)
{
    return _dslContext.Limit(limit);
}
public ISelectAfterLimitOffsetStep Offset(int val)
{
    return _dslContext.Offset(val);
}
public ISelectAfterLimitOffsetStep Offset(string offset)
{
    return _dslContext.Offset(offset);
}
public ISelectHavingConditionStep Having(ConditionBase condition)
{
    return _dslContext.Having(condition);
}
public ISelectHavingConditionStep Having(Condition condition)
{
    return _dslContext.Having(condition);
}
public ISelectHavingConditionStep Having(string condition)
{
    return _dslContext.Having(condition);
}
public ISelectHavingStep GroupBy(params string[] aggregatedItems)
{
    return _dslContext.GroupBy(aggregatedItems);
```

```
        }
        public ISelectHavingStep GroupBy(params IGroupingItem[] groupingItems)
        {
            return _dslContext.GroupBy(groupingItems);
        }
        public ISelectConditionStep Where(string condition)
        {
            return _dslContext.Where(condition);
        }
        public ISelectConditionStep Where(Condition condition)
        {
            return _dslContext.Where(condition);
        }

        public ISelectConditionStep Where(ConditionBase condition)
        {
            return _dslContext.Where(condition);
        }
        public ISelectConditionStep WhereExists(Action<ISelectStep> selectAction)
        {
            return _dslContext.WhereExists(selectAction);
        }
        public ISelectConditionStep WhereExists(ISelectResult select)
        {
            return _dslContext.WhereExists(select);
        }
        public ISelectConditionStep WhereNotExists(Action<ISelectStep> selectAction)
        {
            return _dslContext.WhereNotExists(selectAction);
        }
        public ISelectConditionStep WhereNotExists(ISelectResult select)
        {
            return _dslContext.WhereNotExists(select);
        }
        public ISelectJoinOnStep Join(ITable table)
        {
            return CreateJoinFromTableHelper(table, JoinType.Join);
        }
        public ISelectJoinOnStep RightJoin(ITable table)
        {
            return CreateJoinFromTableHelper(table, JoinType.RightOuterJoin);
        }
        public ISelectJoinOnStep LeftJoin(ITable table)
        {
            return CreateJoinFromTableHelper(table, JoinType.LeftOuterJoin);
        }
        public ISelectJoinOnStep InnerJoin(ITable table)
        {
            return CreateJoinFromTableHelper(table, JoinType.Join);
```

```
}
public ISelectJoinOnStep FullJoin(ITable table)
{
    return CreateJoinFromTableHelper(table, JoinType.FullOuterJoin);
}
public ISelectJoinOnStep Join(string table)
{
    return CreateJoinFromStrTableHelper(table, JoinType.Join);
}
public ISelectJoinOnStep RightJoin(string table)
{
    return CreateJoinFromStrTableHelper(table, JoinType.RightOuterJoin);
}

public ISelectJoinOnStep LeftJoin(string table)
{
    return CreateJoinFromStrTableHelper(table, JoinType.LeftOuterJoin);
}
public ISelectJoinOnStep InnerJoin(string table)
{
    return CreateJoinFromStrTableHelper(table, JoinType.Join);
}
public ISelectJoinOnStep FullJoin(string table)
{
    return CreateJoinFromStrTableHelper(table, JoinType.FullOuterJoin);
}
public ISelectJoinOnStep Join(IName name)
{
    return CreateJoinFromStrTableHelper(name.Name, JoinType.Join);
}
public ISelectJoinOnStep RightJoin(IName name)
{
    return CreateJoinFromStrTableHelper(name.Name, JoinType.RightOuterJoin);
}
public ISelectJoinOnStep LeftJoin(IName name)
{
    return CreateJoinFromStrTableHelper(name.Name, JoinType.LeftOuterJoin);
}
public ISelectJoinOnStep InnerJoin(IName name)
{
    return CreateJoinFromStrTableHelper(name.Name, JoinType.Join);
}
public ISelectJoinOnStep FullJoin(IName name)
{
    return CreateJoinFromStrTableHelper(name.Name, JoinType.FullOuterJoin);
}
public ISelectLimitStep OrderBy(params string[] sortingItems)
{
    return _dslContext.OrderBy(sortingItems);
```

```
        }
        public ISelectLimitStep OrderBy(params IOrderByItem[] orderByItems)
        {
            return _dslContext.OrderBy(orderByItems);
        }
        public IField Field(int index)
        {
            return _dslContext.Field(index);
        }
        public IField Field(string name)
        {
            return _dslContext.Field(name);
        }

        public IField Field(IField field)
        {
            return _dslContext.Field(field);
        }
        public ITable AsTable(string name)
        {
            return _dslContext.AsTable(name);
        }
        public IField AsField(string alias)
        {
            return _dslContext.AsField(alias);
        }
        public DataSource AddToFromClause(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
        {
            return _dslContext.AddToFromClause(queryModifier, unionSubQuery);
        }
        public void AddToSelectList(IQueryModifier queryModifier, UnionSubQuery
unionSubQuery)
        {
            _dslContext.AddToSelectList(queryModifier, unionSubQuery);
        }
        private ISelectJoinOnStep CreateJoinFromStrTableHelper(string table, JoinType
joinType)
        {
            var sqlFromSource =
_dslContext.ActiveUnionSubQuery.SQLContext.ParseDatasource(table);
            var rightDataSource =
_dslContext.QueryModifier.AddDataSource(_dslContext.ActiveUnionSubQuery,
                sqlFromSource, typeof(DataSourceObject));

            return new SelectJoinOnStep(_dslContext, _dataSource, rightDataSource,
joinType);
        }
```

```
        private ISelectJoinOnStep CreateJoinFromTableHelper(ITable table, JoinType
joinType)
        {
            var rightDataSource = table.AddToFromClause(_dslContext.QueryModifier,
_dslContext.ActiveUnionSubQuery);

            return new SelectJoinOnStep(_dslContext, _dataSource, rightDataSource,
joinType);
        }
    }
```

Файл WithAsStep.cs

```
    public class WithAsStep : IWithAsStep
    {
        private readonly IDSLContext _dsl;
        private readonly string _alias;

        public WithAsStep(IDSLContext dsl, string alias)
        {
            _dsl = dsl;

            _alias = alias;
        }

        public IDSLContext As(string select)
        {
            _dsl.Result.AddNewCTE(select, _alias);

            return _dsl;
        }

        public IDSLContext As(Action<ISelectStep> selectAction)
        {
            using (var innerSelect = new DSLContext(_dsl.Result.SQLContext))
            {
                selectAction(innerSelect);
                innerSelect.BeforeGetResult();

                _dsl.Result.AddNewCTE(innerSelect.SQL, _alias);
            }

            return _dsl;
        }

        public IDSLContext As(ISelectResult select)
        {
            return As(select.SQL);
        }
    }
```