Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Создание проблемно-ориентированного языка для управления физическим уровнем баз данных

Дипломная работа студентки 544 группы

Силиной Ольги Александровны

Научный руководитель	.....	к.ф.-м.н., доц. Д.В. Барашев
	/подпись/	
Рецензент	.....	ст. преп. Д.В. Луцив
	/подпись/	
“Допустить к защите”	.....	д.ф.-м.н., проф. А.Н. Терехов
Заведующий кафедрой	/подпись/	

Санкт-Петербург

2010

St. Petersburg State University

Faculty of Mathematics and Mechanics

Chair of Software Engineering

Development of a DSL for database storage level management

Graduate paper of student of group 544

Olga Silina

Academic advisor	.....	associate professor Dmitry Barashev
	/signature/	
Reviewer	.....	assistant professor Dmitry Luciv
	/signature/	
"Admitted to defense"	.....	professor Andrey Terekhov
Chair leader	/signature/	

St. Petersburg

2010

Содержание

1 Введение

Большинство приложений, используемых в современном мире, оперирует различными данными. Проблема хранения данных, необходимых приложению для работы, появилась так же давно, как сами приложения. В разное время для этого применялись разные решения [6].

Для большого количества приложений этот вопрос может быть решен при помощи работы с автономными базами данных, находящимися на независимом сервере и взаимодействующими с основным приложением при помощи специальных библиотек, таких как, например, JDBC (Java Database Connectivity) для языка Java. JDBC – библиотека, предоставляющая интерфейс для соединения с сервером базы данных и выполнения запросов в форме SQL.

Позднее появились встраиваемые системы управления базами данных, такие как BerkleyDB или SQLLite, позволяющие обращаться к данным из приложения при помощи специальных библиотек, скрывающие от разработчика детали взаимодействия с базой и требующие значительно меньше усилий для доступа к данным. Обычно такие системы имеют реализации для разных платформ и языков и предоставляют широкий спектр дополнительных возможностей.

Однако многим современным приложениям не подходит табличная структура хранения информации, поэтому приходится разрабатывать подходы, отличные от реляционной модели. Появляется всё больше специализированных систем хранения информации, часто оптимизированных под конкретные типы хранимых данных. Например, для индексации многомерной информации, такой, как географические данные, применяются такие структуры, как R-деревья. Всё чаще возникает необходимость создания сложных конструкций, отвечающих за хранение информации.

Для реализации этой задачи можно применить несколько подходов. Самый простой вариант – создание библиотеки для управления хранением данных. Недостаток такого подхода в том, что библиотеки специфичны для конкретных платформ и языков, поэтому для каждой используемой платформы придется написать свою библиоте-

ку. Перенос приложения на другую платформу потребует от разработчиков изучения новой библиотеки.

Решением этой проблемы может стать создание проблемно-ориентированного языка. Проблемно-ориентированные языки (Domain Specific Languages) определяются как языки, привязанные к определенной предметной области, разработанные специально для нее и позволяющие выражать понятия, связанные с этой областью, эффективнее, чем языки общего назначения. Такие языки используются для работы на высоком уровне абстракции.

Целью данной работы является создание проблемно-ориентированного языка программирования, предназначенного для работы с физическим уровнем базы данных.

Язык должен быть достаточно простым для изучения, но в то же время достаточно выразительным для того, чтобы с его помощью можно было создавать и настраивать такие структуры как, например, B-деревья.

Такой проблемно-ориентированный язык может также эффективно использоваться в качестве платформы для экспериментов с различными индексными структурами.

В работу по созданию языка входят следующие части: разработка структуры и синтаксиса языка, создание синтаксического анализатора, а также создание транслятора, переводящего конструкции разработанного языка в конструкции языка Java. Такой транслятор позволит также использовать код, написанный на нашем языке, совместно с кодом на других языках, исполняемых на JVM. В будущем также возможно написание трансляторов для других языков.

2 Обзор существующих решений

Ниже приведен обзор известных автору подходов к решению проблемы низкоуровневой организации хранения данных.

Первым языком, позволяющим удобно описывать структурированные данные, отображаемые в физические файлы, был COBOL. В COBOL встроены возможности для работы с файлами, состоящими из типизированных записей, как с последователь-

ным, так и с прямым доступом.

Один из подходов к решению проблемы в других языках – расширение уже существующего языка общего назначения при помощи конструкций, отвечающих за операции, связанные с хранением данных. Этот подход, реализованный, например, в языке E [11], имеет некоторые недостатки. Прежде всего, так как E – это расширение конкретного языка (C++), он недостаточно общий, и для каждого нового целевого языка необходимо будет реализовать свое расширение, учитывая специфику синтаксиса и возможности целевого языка. Основной задачей, решаемой при создании языка E, является разработка прозрачного синтаксиса для работы с постоянно хранимыми переменными. При этом физическое расположение объектов на диске задается на уровне механизма хранения данных, независимого от языка. Практически, программист может только передавать на более низкий уровень некоторые метки, которые используемая база данных использует в зависимости от её реализации (возможно, игнорирует вообще). Несмотря на это, E – мощный язык, поддерживающий, например, возможность транзакций.

Система управления хранением объектов, реализованная в проекте SHORE [2], дополняет функциональность, реализованную в EXODUS (системе управления хранением объектов, использовавшейся в E [4]). Целью проекта SHORE является объединение объектно-ориентированной базы данных с имеющимися технологиями в области файловых систем. SHORE отказывается от использовавшейся в EXODUS клиент-серверной модели в пользу симметричной структуры. В итоге мы имеем более легковесную и простую в понимании и использовании схему. SHORE так же позволяет использовать базу данных, написанную на одном языке, из приложений, написанных на других языках, что обеспечивает лучшую переносимость.

Ещё один подход к хранению данных при помощи файлов, проецируемых в основную память (memory-mapped files) предложен в системе управления хранением объектов Dali [8]. Этот подход фокусируется на достижении быстродействия и подходит для применения в приложениях, требующих очень быстрой обработки запросов. При этом база данных рассматривается как совокупность файлов данных, каждый из

которых может быть отдельно спроецирован в основную память. Подразумевается, что большая часть данных в каждый момент времени находится в основной памяти, поэтому использование этой системы возможно только при наличии очень большого количества свободной памяти или сложного и очень эффективного механизма кеширования.

Подробный обзор области языков для хранения данных и объектно-ориентированных языков дается в работе [3]. В ней дается критика тяжелых клиент-серверных систем управления хранения объектов. Добавляя в приложение дополнительный уровень опосредованности и требуя от разработчика специальных знаний и опыта работы с аналогичными системами, такие приложения не оправдывают себя на реальных задачах и, следовательно, редко используются на практике. Они также не дают контроля над некоторыми низкоуровневыми деталями, такими как буферизация или паралеллизм.

Ещё один разносторонний обзор взаимодействия языков программирования и баз данных рассматривается в статье [5], в которой предпринята попытка сформулировать критерии оценки эффективности существующих решений и основные проблемы, встающие перед разработчиком в данной области.

3 Описание языка

Основной задачей языка является абстрагирование работы с двоичными файлами, а именно:

- поблочная итерация
- управление буферами
- управление временными файлами
- освобождение ресурсов
- преобразование двоичных данных в структурированный вид
- сравнения структурированных данных

Язык позволяет манипулировать следующими сущностями:

- пул файлов
- файл
- временный файл
- итератор для чтения из файла
- поток для записи в файл
- объекты, хранимые по ссылке
- объекты, хранимые по значению
- значения примитивных типов
- массивы
- записи

- списки переменной длины

В исполняемом коде используются следующие конструкции:

- последовательная композиция
- условный оператор
- циклы с пред- и постусловием
- циклы с итератором
- структурные переходы break и continue
- вызовы процедур и функций
- присваивание переменных + система типов

Декларации, встречающиеся в языке:

- импорт классов целевого языка
- объявления структур данных, хранимых по значению
- объявления глобальных переменных
- объявления методов

Управляющие конструкции имеют синтаксис и семантику, аналогичную управляющим конструкциям языка Java.

3.1 Пулы файлов

Для работы с данными требуется размещать их в файловой системе. Поскольку необходимо абстрагировать организацию файлов от конкретной реализации, зависящей от файловой системы, системы прав доступа и других факторов, вводится понятие пула файлов.

В программе вводится пул файлов по умолчанию (пул верхнего уровня). Для обозначения пула файлов верхнего уровня используется следующий ключевое слово root:

```
filepool root
```

Для моделирования иерархической файловой системы и логического разделения файлов, предназначенных для разных целей, вводится возможность создавать именованные пулы файлов, вложенные друг в друга.

```
filepool inner_pool in pool name ‘‘my_inner_files’’
```

Такая конструкция позволяет абстрагировать идею вложенных папок или групп файлов с одинаковым префиксом.

Гарантируется, что файлы, созданные в разных подпулах или с разными именами, будут существовать на диске независимо. Конкретные детали расположения файлов не оговариваются и зависят от реализации на целевом языке.

Для работы с временными файлами используется специальный пул `temp`.

Для обхода иерархии имеющихся в программе файлов есть возможность перебрать все подпулы данного пула при помощи следующей конструкции:

```
filepools of pool
```

Эта конструкция выдает список всех пулов, созданных внутри данного пула, или пустой список в случае, если пул не имеет вложенных пулов.

Аналогично для получения списка всех файлов в данном файловом пуле используется похожая конструкция:

```
files of pool
```

3.2 Файлы

При операциях с дисковым пространством используются файлы, создаваемые и управляемые внутри файловой системы. Внутри программы для обращения к файлу используются файловые переменные. Каждой такой переменной при создании ставится в соответствие определенный файл файловой системы, который остается привязан к ней на протяжении всего времени жизни переменной. Любое взаимодействие с фай-

ловой системой происходит посредством файловой переменной.

При создании нового файла указывается его имя и пул, в котором он будет размещен.

Для создания файла в заданном пуле используется следующий синтаксис:

```
file ex in pool name ‘‘example_file’’
```

Для проверки существования файла на диске используется встроенная функция `exists`, возвращающая `true`, если файл с данным именем существует и `false` в обратном случае:

```
exampleFile.exists
```

3.2.1 Временные файлы

В языке предоставляются возможности для дополнительного абстрагирования при работе с временными файлами. Временные файлы создаются в особом пуле временных файлов. При создании временного файла указывать его имя не требуется. Именование файлов и размещение их на диске осуществляется для временных файлов автоматически, при этом гарантируется, что дисковое пространство, соответствующее временному файлу не будет соответствовать никакому другому файлу.

При окончании работы с файлом он закрывается и все изменения записываются из буферов на диск. Это может происходить как автоматически, так и по явно. Автоматическое освобождение файла происходит при выхода соответствующей файловой переменной из области видимости. Явное освобождение файла происходит при помощи вызова на соответствующей файловой переменной функции release. Вызов имеет следующий синтаксис:

```
exampleFile.release
```

При освобождении временного файла он также удаляется с диска, что позволяет разработчику избавиться от дополнительных усилий по освобождению неиспользуемого места на диске.

Удаление постоянных файлов возможно явным образом при помощи вызова функ-

ции `delete`. Удаление возможно только для закрытых файлов, при попытке удаления открытого файла происходит ошибка времени выполнения.

3.3 Итераторы

Для абстрагирования чтения из двоичного файла используются итераторы. Основная задача итератора состоит в том, чтобы производить последовательное чтение из файла при помощи буфера, находящегося в оперативной памяти. При этом ставится цель максимально избавить разработчика от низкоуровневых деталей реализации, оставив, однако, полезные возможности для контроля за тем, как происходит чтение.

Для итерации по записям, хранящимся в файле, необходимо указать их тип. При создании итератора с файлом ассоциируется буфер в оперативной памяти заданного размера. По мере продвижения по файлу в буфер считываются новые порции записей.

Для того, чтобы начать итерацию по файлу, считывая данные порциями заданного размера, используется следующая конструкция:

```
for (buffer :: inputfile by Data(100)) {
    ...
}
```

Итератор, созданный для этого цикла, будет продвигаться по файлу, считывая в буфер в оперативной памяти записи типа Data порциями по 100 записей. Если в файле осталось менее 100 записей, будет прочитано столько записей, сколько осталось в файле.

Тип данных, используемый в подобных конструкциях, должен быть объявлен в программе способом, который будет описан позднее.

Такой итератор существует только внутри цикла, в котором он создан, и будет отпущен при завершении цикла.

В случае, когда размер буфера для итерации не важен для разработчика, он может опустить указание на размер. В таком случае оптимальный размер буфера будет вычислен автоматически в зависимости от реализации на целевом языке.

При необходимости передавать итератор в другую часть программы или обращаться к нему по имени возможно явное создание итератора. Для этой цели используется следующий синтаксис:

```
i = iterator of Data for inputfile
```

По умолчанию итератор при создании указывает на первую запись файла. Возможно создание итератора, указывающего на запись в другом месте файла. Это осуществляется при помощи следующей конструкции:

```
i = iterator of Data for inputfile startingat 100
```

При создании итератора, привязанного к файлу, файл автоматически открывается на чтение. По завершении прохода итератора по файлу файл отпускается и закрывается. Если такое поведение нежелательно, итератор можно пометить при помощи ключевого слова norelease.

```
i = iterator of Data for inputfile norelease
```

Завершение итерации при помощи такого итератора не будет приводить к отпусканию файла.

При необходимости завершить итерацию по файлу явным образом до окончания прохода итератора итератор можно отпустить:

```
i.release
```

При этом освобождается оперативная память, занятая под буфер итератора, и отпускается читаемый при помощи итератора файл.

Для определение текущего состояния итератора используется вызов на итераторе функции `released`, возвращающей `true`, если итератор отпущен, и `false` в обратном случае.

Для доступа к данным, читаемым при помощи итератора используются функции `hasData`, `data` и `next`.

Вызов функции `hasData` возвращает `true` в случае, если итерация не завершена и, следовательно, итератор указывает на запись файла, и `false`, если итерация завершена и итератор отпущен.

Вызов функции `data` возвращает запись, на которую в данный момент указыва-

ет итератор, если она есть, и вызывает ошибку времени выполнения, если итерация завершена и итератор отпущен.

Вызов функции `next` передвигает итератор на одну запись вперед в случае, если итерация не завершена, и вызывает ошибку времени выполнения, если итерация завершена и оператор отпущен.

Для дискового файла возможно создание неограниченного количества итераторов.

Попытка создать итератор для файловой переменной, файл для которой не существует, приведет к ошибке времени выполнения.

3.4 Потоки для записи в файл

Для абстрагирования процесса записи в двоичный файл применяются потоки записи. Внутри потока происходит буферизованная запись на диск. Запись в поток происходит внутри блока записи. Чтобы открыть блок записи, используется следующая конструкция:

```
writing Data to outputFile {
    ...
}
```

В начале такого блока происходит открытие файла на запись. Открыть файл на запись можно одновременно не более одного раза, поэтому в случае, если файл уже был открыт на запись ранее, произойдет ошибка времени выполнения.

Запрещается открывать файл одновременно на чтение и на запись. В случае, если попытаться создать итератор по файлу, открытому на запись, или попытаться открыть на запись файл, для которого существует неотпущенный итератор, произойдет ошибка времени выполнения.

После окончания выполнения кода, находящегося внутри блока записи, буфер записи автоматически записывается на диск, если в нем есть незаписанные данные, и файл закрывается.

3.5 Объекты, хранимые по ссылке

В случае, когда функциональности языка не хватает для реализации требуемого алгоритма, разработчик может включать в код фрагменты, реализованные на целевом языке. Для использования объекта целевого языка необходимо предварительно объявить внешний объект, указав, какому классу целевого языка он соответствует. Декларация внешнего объекта имеет следующий синтаксис:

```
object example : org.example.ExampleObject
```

После объявления объекта разрешается вызывать его методы, передавая соответствующие параметры и используя возвращаемые методами значения по мере надобности. При этом проверка типов происходит уже на уровне сгенерированного кода, поэтому при написании когда, использующего объекты целевого языка, необходимо заботиться о том, чтобы типы совпадали.

Любой код, соблюдающий это ограничение, считается допустимым. Так, например, следующий код корректен в случае, если объект, переданный под именем `iterable`, реализует интерфейс, позволяющий итерацию.

```
method iterate(object iterable) {
    for (object i : iterable) {
        ...
     }
}
```

3.6 Объекты, хранимые по значению

3.6.1 Примитивные типы

Поддерживаемые примитивные типы соответствуют типам, поддерживаемым в языке Java. Поддерживаются следующие примитивные типы:

- булевский тип boolean

- целочисленные типы
  - byte
  - short
  - int
  - long
- вещественные типы
  - float
  - double

Размер каждого типа и способ представления в памяти соответствуют соответствующим размерам и способам представления из системы типов языка Java.

3.6.2 Массивы

Для хранения наборов значений одного типа фиксированной длины поддерживаются массивы.

При объявлении массива обязательно должна указываться его длина:

```
int[25] integers
```

Индексами массива могут являться целые числа, индексация начинается с нуля. Для обращения к элементу массива используется стандартный для многих языков синтаксис:

```
integers[5]
```

При попытке обращения по индексу к элементу массива, которого не существует, произойдет ошибка времени выполнения.

Для того, чтобы узнать размер массива, используется функция `size`:

```
intergers.size
```

3.6.3 Записи

Для абстракции сложных структур данных используются записи. Запись — это совокупность одного или более полей, каждое из которых может иметь в качестве значения значение примитивного типа, массив значений примитивного типа или другую запись.

Объявление записи имеет следующий синтаксис:

```
rec Data {
    key : int
    value : char[250]
    x : rec {
        ...
    }
}
```

Запись является основной единицей хранения больших объемов данных.

По объявлению записи вычисляется место, занимаемое ей на диске: суммируются размеры, занимаемые каждым из полей записи, если поле записи само является записью, то размер считается рекурсивно: сначала вычисляется размер внутренней записи.

Порядок хранения элементов записи на диске соответствует порядку объявления полей.

Для доступа к полю записи используется обращение по имени:

```
myrec.key
```

3.7 Списки переменной длины

Для хранения наборов значений переменной длины используются списки. Для объявления списка используется ключевое слово `list`:

```
list items
```

При инициализации список создается пустым.

Работа со списками поддерживает следующие операции:

- добавление элемента в конец
- итерация по списку
- проверка на пустоту
- выяснение длины списка

Добавление элемента в конец происходит при помощи конструкции `+=`:

```
items += item
```

При осуществлении итерации по списку его элементы перебираются в цикле по одному от начала к концу:

```
for (item : items) {
    ...
}
```

Проверка списка на пустоту происходит при помощи функции `empty`, возвращающей `true` в случае, если список пуст, и `false` в обратном случае:

```
items.empty
```

Для выяснения длины списка используется функция `size`, возвращающая текущее количество элементов в списке.

3.8 Особенности генерации

Из написанного кода генерируется код на целевом языке. Для каждой программы создаются следующие классы:

- основной класс программы
- классы данных

При написании программы разработчик организовывает код в методы и поля, из которых при генерации создаются методы и поля основного класса.

Для объявления поля используется ключевое слово `field`.

`field int 5 field org.example.Tokenizer tokenizer`

Поле может хранить значение примитивного типа, массив значений примитивного типа, список значений примитивного типа или объект целевого языка и будет транслировано в соответствующий объект целевого языка.

Для вызова сгенерированного кода из внешнего приложения используются его методы. Код организовывается в методы при помощи ключевого слова `method`.

```
method search(filepool sorted, char[] query) {
    ...
}
```

В такой метод в качестве параметров передается информация, требующаяся коду для работы:

- указатели на данные, используемые в программе
- компараторы, требующиеся для сравнения записей
- объекты целевого языка, требуемые программе для работы

При генерации из метода создается публичный метод основного класса с соответствующей сигнатурой.

Для каждого класса записи генерируется класс, хранящий внутри себя байтовый массив. Для каждого поля записи создаются методы, выдающие объект, соответствующий этой записи, возвращающие в зависимости от типа записи примитивный тип, массив значений примитивного типа или объект другой записи. Также для каждого поля записи создаются методы, позволяющие задавать величину каждого поля и соответственно модифицирующие внутренний байтовый мотив.

3.9 Встроенные вспомогательные операции

Операции сравнения, поиска минимума и максимума и сортировки являются базовыми и очень важными операциями для обработки данных, поэтому для них в языке реализованы особые вспомогательные операции.

Реализованные операции сравнения включают в себя:

- `>`
- `<`
- `==`
- `<>`
- `<=`
- `>=`

Для сравнения данных используются компараторы. Компаратор — объект целевого языка, предназначенный для инкапсуляции знаний о том, как сравнивать объекты определенного типа.

Компараторы передаются в программу в качестве параметров методов и обозначаются ключевым словом comparator.

```
method (comparator comp) {
    ...
}
```

Для каждого из примитивных численных типов имеется свой компаратор по умолчанию. Для получения объекта такого компаратора используется следующий синтаксис:

```
int.comparator
```

При сравнении записей используется синтаксис, похожий на сравнение данных примитивных типов:

```
rec1 <comp< rec2
```

При сравнении примитивных типов при помощи компаратора по умолчанию можно опускать имя компаратора и использовать сокращенную запись в привычной форме:

```
1 == 2
```

Для нахождения минимума/максимума в массиве данных могут использоваться встроенные функции min/max. Эти функции принимают на вход массив с данными, тип записей, хранящихся в массиве, и компаратор для сравнения этих записей, и возвращают минимальную/максимальную запись, сравнивая записи при помощи переданного компаратора.

```
min(buffer, Data, comparator)
```

Для сортировки массива данных используется встроенная процедура sort, также принимающая на вход массив данных, тип записей, находящихся в массиве и компаратор для сравнения этих записей. После применения процедуры в массиве оказываются записи, отсортированные по возрастанию.

4 Пример использования

В предыдущих частях приводилось описание синтаксиса и семантики языка, предназначенного для абстракции работы с дисковыми данными. В этой части приведено несколько примеров кода на разработанном языке.

В качестве примера используется реализация алгоритма сортировки многоканальным слиянием.

```
method mergeSort(file inputFile, file outputFile,
        comparator dataComparator, type Data, int BUFFER_SIZE_IN_RECORDS) {
    list tempIterators
    for (buffer :: inputFile by Data(BUFFER_SIZE_IN_RECORDS)) {
        sort(buffer, Data, dataComparator)
        file tempfile in temp // such files are deleted on release
        auto tempfile <- buffer
        tempIterators += iterator of Data for tempfile
     }

     writing Data to outputFile {
         do {
             min = findMinAndPull(tempIterators, dataComparator)
             if (min != null) {
                 auto outputFile <- min
             } else {
                 break;
             }
         } while (true)
     }
}
```

```
method findMinAndPull(list tempIterators, comparator comp) {
    min = null
    for (i : tempIterators) {
        if (!i.released || i.hasData) {
          if (min == null || min >comp> i.data) {
            min = i.data
            i.next
          }
        } else {
            release i
        }
     }
    return min
}
```

Как продолжение этого примера, можно привести пример простейшего поискового инструмента, строящего индекс по данным ему документам и осуществляющего поиск по ним.

```
method index(filepool pool, object corpus) {
    list tokens
    for (object doc : corpus) {
    object tokenizer : org.example.TokenizerFactory
    for (token :: tokenizer.tokenize(doc.contents)) {
        file tokenFile in pool name token
        if (!tokenFile.exists) {
            tokens += token
        }
        auto tokenFile <- doc.id
    }
```

```
    filepool unsorted in pool name ‘‘unsorted’’
    filepool sorted in pool name ‘‘sorted’’
    for (token : tokens) {
        file uns in unsorted name token
        file s in sorted name token
        mergeSort(uns, s, int.comparator, int)
    }
}

field org.example.Tokenizer tokenizer

method search(filepool sorted, char[] query) {
    list tokenFiles;
    for (token :: tokenizer.tokenize(query)) {
        tokenFiles += iterator of int for tempfile
    }
    list docIds;
    while (true) {
        v = getSameValueFromTops(int, tokenFiles, int.comparator);
        if (v == null) {
        min = findMinAndPull(tokenFiles, int.comparator)
        if (min == null) {
            break;
        } else {
            docIds += v;
            pullAll(tokenFiles);
        }
    }
```

```
    return docIds
}

method pullAll(list iters) {
    for (i : iters) {
        if (i.hasData) {
            i.next
        }
    }
}

method getSameValueFromTops(type Data, list iterators, comparator comp) : Data {
    Data top = null
    for (i : iterators) {
        if (i.hasData) {
            if (top == null) {
                top = i.data
            } else if (top !comp= i.data) {
                return null;
            }
        }
    }
    return top;
}
```

5 Программная реализация

Языком реализации был выбран Haskell. Haskell – лаконичный язык с сильной системой типов и строгой формальной семантикой. Система типов Haskell позволяет отлавливать большую часть ошибок программиста в момент компиляции, что позволяет писать более надежный и заслуживающий доверия код. Код, написанный на Хаскелле, несколько проигрывает в производительности коду, написанному на C/C++, однако этот недостаток компенсируется преимуществами в простоте и скорости разработки и надежности [10].

При написании синтаксического анализатора на Haskell существуют две альтернативы: использование библиотеки комбинаторов синтаксических анализаторов или использование автоматического генератора синтаксических анализаторов.

Комбинаторы синтаксических анализаторов используются для написания базовых синтаксических анализаторов и конструирования при их помощи более сложных синтаксических анализаторов, распознающих правила, определяющие нетерминалы. Самой широко распространенной такой библиотекой для Haskell является библиотека Parsec, позволяющая создавать быстрые комбинаторы синтаксических анализаторов с использованием монад.

Другая альтернатива, которая и была выбрана для реализации синтаксического анализа в данной работе – это использование системы генерации синтаксических анализаторов Happy [9].

Happy – система, позволяющая автоматически генерировать синтаксические анализаторы нсинтаксические анализаторы Haskell, похожая на многим известный ‘yacc‘ [1] для C. Как и yacc, она принимает на вход файл, содержащий аннотированную BNF-спецификацию грамматики и создаёт модуль на Haskell, содержащий синтаксический анализатор этой грамматики.

Happy – гибкая система, позволяющая иметь в одной и той же программе несколько синтаксических анализаторов и несколько начальных нетерминалов для одной и той же грамматики. Happy может как работать в сочетании с лексическим анализа-

тором, предоставленным пользователем (написанным вручную или сгенерированным при помощи какой-то другой программы), так и разбирать поток символов напрямую.

Парсеры, созданные при помощи Happy, легки в реализации, просты для понимания и расширения и выигрывают в производительности у эквивалентных синтаксических анализаторов, написанных при помощи комбинаторов синтаксических анализаторов или аналогичных инструментов.

Для создания лексического анализатора использовался Alex – инструмент для генерации лексических анализаторов на Haskell [7]. Alex принимает на вход описание токенов, которые он должен распознавать, в форме регулярных выражений. Он похож на lex/flex для C/C++. Alex полностью совместим с Happy, и обладает лаконичным и понятным синтаксисом.

6 Заключение

В работе предложен подход к проектированию и реализации работы с физическим уровнем хранения данных. В основе подхода лежит создание проблемно-специфичного языка, код на котором впоследствии генерируется в код на целевом языке.

В качестве основных концепций, для которых разработаны конструкции, упрощающие написание кода, связанного с манипуляцией данными на диске и инкапсулирующие широко встречающиеся в таком коде абстракции, используются поблочная буферизованная итерация, управление буферами в оперативной памяти, буферизованная запись в файл и сравнение структурированных данных, а также работа с файлами и временными файлами.

Разработана реализация лексического и синтаксического анализаторов для созданного языка на языке Haskell при помощи инструмента для генерации лексических анализаторов Alex и инструмента для генерации синтаксических анализаторов Happy, а также транслятор, переводящий код на разработанном языке в код на языке Java.

Для иллюстрации возможностей языка реализован пример кода, осуществляющего многоканальную сортировку слиянием и строящего с её помощью простейший индекс для поиска по набору документов. Пример наглядно демонстрирует лаконичность и легкочитаемость кода на разработанном языке по сравнению с аналогичным кодом, использующим, например, Java-библиотеки.

Одно из направлений дальнейших исследований заключается в реализации трансляторов из разработанного языка в другие языки программирования, часто используемые в приложениях, работающих с физическим уровнем организации баз данных (например, C++). Другое направление может заключаться в расширении имеющегося синтаксиса языка с целью добавления новых возможностей.

Список литературы

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006.

[2] M. Carey and D DeWitt. Shoring up persistent applications. *ACM SIGMOD Record*, 23:383–394, 1994.

[3] M. Carey and D DeWitt. Of objects and databases: a decade of turmoil. *Proceedings of the international conference on very large databases*, 22:3–15, 1996.

[4] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. *Store management for objects in EXODUS*. ACM New York, NY, USA, 1989.

[5] W. Cook and A Ibrahim. Intergating programming languages and databases: what is the problem?

[6] C. Date. *An introduction to database systems*. Addison-Wesley Reading, MA, 2000.

[7] C. Dornan, I. Jones, and S. Marlow. Alex User Guide. 2003. `http://www.haskell.org/alex/doc/html/index.html`.

[8] H. Jagadish, D Lieuwen, R. Rastogi, and A. Silberschatz. Dali: A High Performance Main Memory Storage Manager. *Proceedings of the International Conference on Very Large Data Bases*, 1994.

[9] S. Marlow. Happy User Guide. 2001. `http://www.haskell.org/happy/doc/html/index.html`.

[10] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, 2009.

[11] J. Richardson and M. Carey. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15:494–534, 1993.