Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Самарин Алексей Владимирович

Исполнитель запросов в системе обработки данных из разнородных источников

Дипломная работа

Допущена к защите.

Зав. кафедрой:

д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:

д. ф.-м. н., профессор Новиков Б. А.

Рецензент:

ассистент Чернышев Г. А.

Санкт-Петербург

2015

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering Chair

Aleksei Samarin

A query executor over heterogeneous information resources

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
professor Boris Novikov

Reviewer:
assistant professor George Chernishev

Saint-Petersburg

2015

Оглавление

Введение

В настоящее время наблюдается стремительный рост объемов информации, хранимой на электронных носителях (см. рис. 1). При этом данные могут быть структурированными (базы данных), неструктурированными (текстовые файлы, веб-сайты, и др.), слабоструктурированными (тексты, и др.).

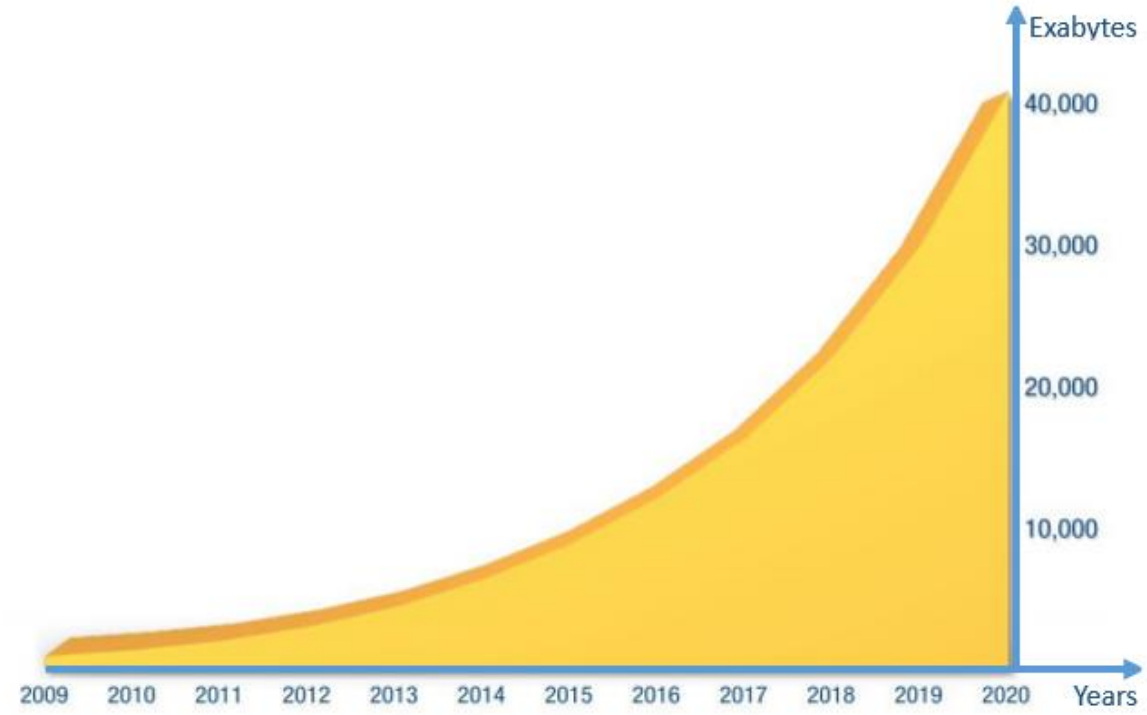

Рис. 1: График роста объема хранимых данных [3].

Помимо всего многообразия и вариативности форматов, используемых при работе с информацией, следует отметить, что сами базы данных могут быть разделены на множество групп по различным признакам. Например, по форматам внутреннего представления данных:

- с поддержкой реляционной модели (PostgreSQL, Oracle DB, MySQL и др.),
- NoSQL базы данных, (MongoDB, CouchDB, Redis, AsterixDB и др.).

Базы данных можно также классифицировать по предоставляемыми интерфейсам для доступа к информации: ODBC, JDBC, ADO и др. Следует также отметить, что одна база может предоставлять сразу несколько видов доступа и управления хранимыми данными.

В рамках различных задач может потребоваться централизованный доступ к разнородным информационным ресурсам. За время своего функционирования многие существующие базы данных успели накопить достаточно большой объем информации для того, чтобы сделать перенос данных в новое централизованное хранилище длительной и трудоемкой задачей.

Подходы, основанные на обработке данных из источников, предоставляющих доступ к информации по какому-либо распространенному интерфейсу, подобные тому, что используется в системе для составления отчетов Pentaho [18], сужают диапазон поддерживаемых информационных ресурсов до баз данных, предоставляющих доступ по поддерживаемому интерфейсу.

Решением данной проблемы может быть создание надстройки над базами данных, обеспечивающей централизованный доступ и управление информацией из уже существующих и функционирующих источников. Основные принципы создания подобных систем представлены в [10]. Также существуют шаблоны проектирования для организации доступа к гетерогенным информационным ресурсам, один из которых представлен в [19].

Разработка инструментов данного типа связана с такими проблемами, как передача подзапросов источникам, выбор алгоритмов для манипулирования данными, организация конвейерной обработки данных [2] из различных источников. Следует также отметить, что зачастую, в рамках рассматриваемых задач требуется поддержка приближенного выполнения и обработки нечетких запросов.

Согласно [7], процесс обработки запросов к различным источникам информации может быть логически разделён на следующие стадии: анализ запроса, оптимизация с целью повышения производительности и исполнение.

Таким образом, система обработки и управления информацией из различных источников может быть разделена на ряд следующих функциональных составляющих.

В первую очередь, можно выделить языковые инструменты для описания нечетких запросов. Данная компонента должна преобразовать языковое представление запроса в древовидную структуру, содержащую операции из расширенной реляционной алгебры [10].

Следующей составляющей является оптимизатор запросов, который получает древовидное представление запроса от языковой компоненты, и производит оптимизации (не меняя тип представления запроса) с целью повышения производительности, основываясь на моделях стоимости исполнения различных операций для каждого исполнителя или адаптера источника.

Еще одной компонентой системы должно быть ядро исполнения запросов, которое получает оптимизированный запрос от оптимизатора, собирает необходимые данные из источников, и производит над ними операции, согласно плану выполнения запроса, полученного от оптимизатора.

В рамках проекта, разрабатываемого на кафедре информационно-аналитических систем математико-механического факультета СПбГУ, частью которого является данная работа, реализуется система управления данными из разнородных информационных ресурсов, основанная на принципах, изложенных в [10], и включающая в себя вышеупомянутые компоненты.

Постановка задачи

Целью данной работы является реализация исполнителя запросов для системы обработки данных из разнородных источников, поддерживающего возможность исполнения операций расширенной реляционной алгебры [10] над информацией, извлеченной из неоднородной среды. В связи с этим были поставлены следующие задачи:

- разработать архитектуру ядра исполнения запросов;
- обеспечить поддержку конвейерной обработки данных в рамках разрабатываемой архитектуры;
- обеспечить возможность интеграции с языковыми компонентами и оптимизатором;
- обеспечить возможность интеграции новых исполнителей и адаптеров источников;
- произвести апробацию разработанной архитектуры при взаимодействии с реализованным ранее исполнителем/источником данных на основе AsterixDB [13] и высокоуровневыми языковыми средствами для описания запросов.

1. Обзор подходов к обработке гетерогенной информации

1.1. Программные интерфейсы для доступа к базам данных

Для унификации доступа к базам данных, существуют программные интерфейсы, такие как ODBC, JDBC, ADO, и др. В рамках текущей задачи, для уменьшения трудоемкости подключения новых источников данных, наиболее оптимальным решением является возможность поддержки любого из вышеперечисленных интерфейсов без изменения архитектуры ядра исполнения.

1.2. Централизованная обработка данных в проекте FORWARD

Одним из проектов, в которых обеспечивается централизованный доступ к различным информационным ресурсам, является FORWARD [11]. В качестве источников данных в этом проекте используются базы данных, поддерживающие собственный язык запросов, такие как Hive, Jaql, Pig, Cassandra, JSONiq, MongoDB, Couchbase, SQL, AsterixDB, BigQuery и UnityJDBC [11]. Таким образом, происходит трансляция из языка запросов SQL++ в языки, использующиеся конкретными базами данных, посредствам адаптеров и специальной программной прослойки (рис. 2). Трансляция производится в языки, использующие либо реляционные модели, либо JSON [11]. С языками запросов, в которые поддерживается трансляция в FORWARD, можно ознакомиться в табл. 1.

В рамках текущей работы реализуется подход, позволяющий обрабатывать данные не только из структурированных источников, поддерживающих свой собственный язык запросов и определённые модели представления данных, но и из неструктурированных источников (таких, как файлы, вебсайты, и др.).

	Language	Version
A	Hive	0.10.0
B	Jaql	0.5.1
C	Pig	0.11.0
D	CQL (Cassandra)	3.1.5
E	JSONiq	1.0.11
F	MongoDB	2.4.7
G	N1QL (Couchbase)	dev.preview 2
H	SQL	SQL:2011
I	AQL (AsterixDB)	0.8.3 beta
J	BigQuery	Released Mar 25 2014
K	Mongo JDBC	4.2.316

Табл. 1: Базы данных, поддерживаемые в SQL++ [11].

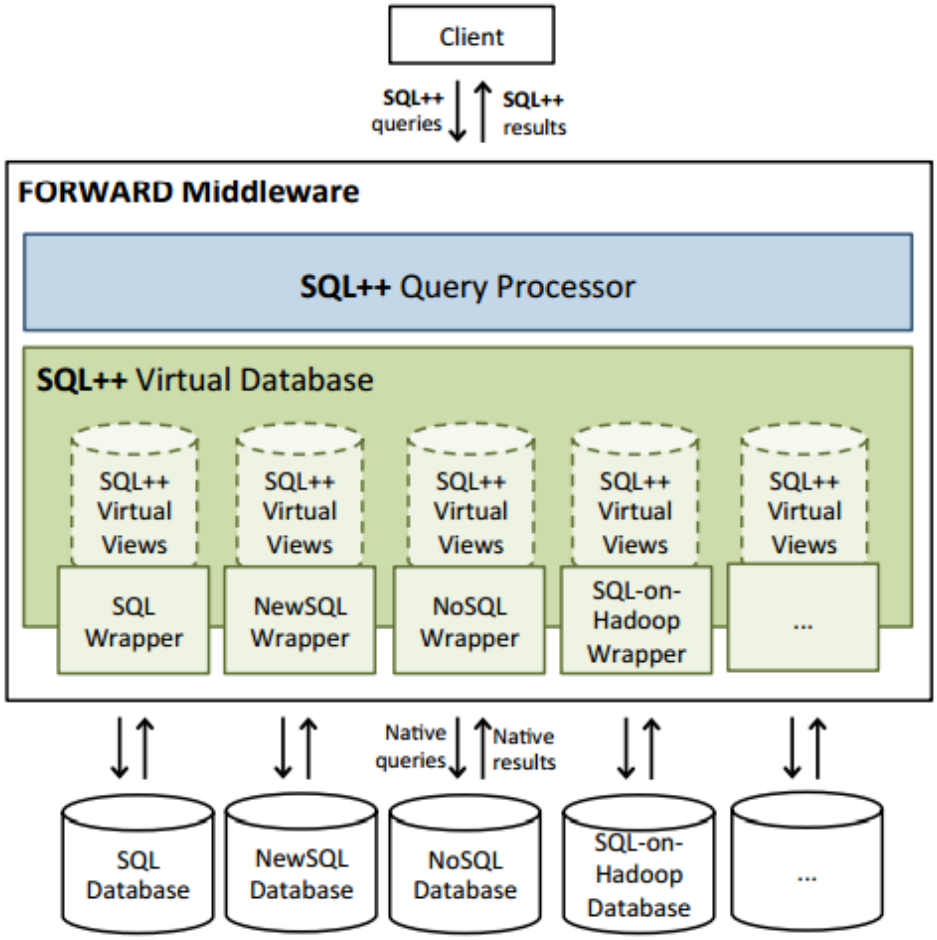

Рис. 2: Архитектура SQL++ [11].

1.3. Паттерн доступа к гетерогенным данным (DAMP)

В статье [19] представлен подход к организации доступа к источникам информации в неоднородной среде в контексте операций *CRUD (copy, retrieve, update, delete)*. В данной статье представлен шаблон проектирования *DAMP*

(*Data Access Management Pattern*) и его реализация на языке C++. Данный паттерн представляет собой описание сервиса для работы с разнородными информационными ресурсами, основными компонентами которого являются сессии (*Sessions*), предоставляющие пользователю интерфейс для работы с сервисом, и посредники (*Mediators*), представляющие собой обертки для работы с конкретными источниками. В рамках данного паттерна, пользователь сам определяет формат запрашиваемых данных, с помощью реализации класса *SessionData* (см. рис. 3).

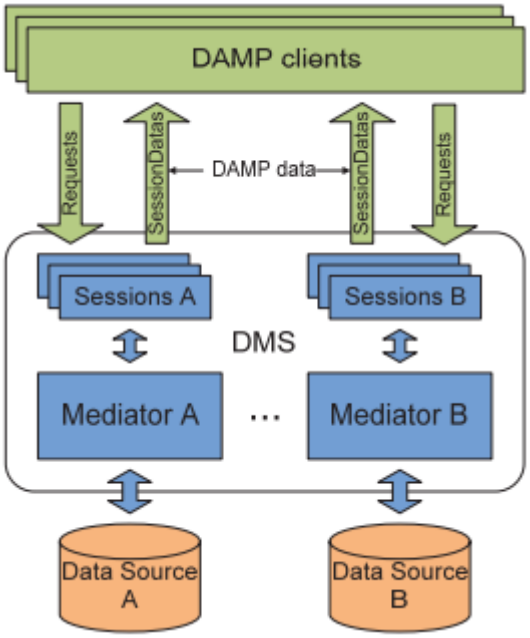

Рис. 3: Архитектура системы DAMP [19].

Концепция использования адаптеров для специфических источников и приведения данных к унифицированному формату используется и в данной дипломной работе, но упомянутый ранее паттерн прежде всего ориентирован на выполнение операций *CRUD*, в то время как данная работа сосредоточена на выполнении операций расширенной реляционной алгебры [10].

1.4. Проект Pentaho

Еще одним примером системы обработки данных из различных источников является программное обеспечение для бизнес-анализа Pentaho. В рамках данного проекта разрабатывается средство для составления отчетов

при использовании OLAP подхода. В качестве одного из средств управления системой предоставляется высокоуровневый язык MDX для описания запросов. Выполнение подобных запросов на источниках является очень длительным процессом, поэтому для обеспечения возможности быстрого выполнения аналитических операций используется ETL модуль, который в результате своей работы производит *data warehouse* [1]. Следует также отметить тот факт, что Pentaho поддерживает интеграцию с базами данных, поддерживающими интерфейс JDBC [16], в то время как в рамках данной работы планируется поддержка обработки информации из гетерогенных источников.

1.5. Модель обработки информации в неоднородной среде на основе Q-set и расширенной реляционной алгебры

Система, частью которой является разрабатываемое в рамках данной работы ядро исполнения запросов, основана на принципах обработки информации из разнородных источников, представленных в статье [10]. Данный подход ориентирован в том числе и на возможность обработки нечетких запросов. Одними, из основных концепций, представленных в [10], являются *q-set* - форма представления запроса в виде тройки: запрос, множество объектов, функция для оценки релевантности объектов; и расширенная реляционная алгебра, состоящая из следующих операций: *filter, fusion, join, aggregation, nest, unnest, group join* [10].

1.6. Принципы построения систем обработки данных

В статье [10] описаны общие детали, касающиеся представления информации и операций для их обработки. Для реализации данных концепций требуются идеи и принципы построения инструментов для обработки разнородной информации, представленные в [9]. Согласно данной статье возможна декомпозиция инструмента обработки данных на такие

функциональные составляющие, как *средства описания запросов, средства оптимизации запросов и ядро исполнения запросов* (см. рис. 4).

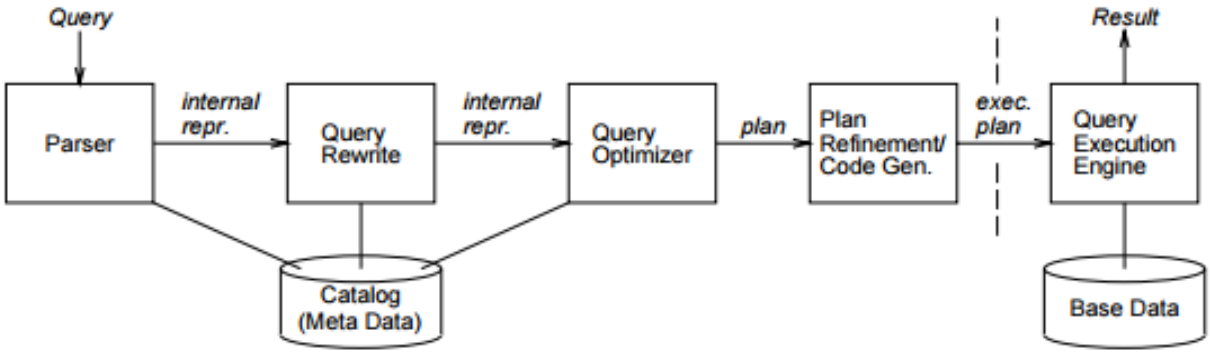

Рис. 4: Процесс обработки данных, и его составляющие [9].

Также в данной статье описан подход к обработке информации из гетерогенных ресурсов на основе посредника для преобразования данных к общему формату (см. рис. 5).

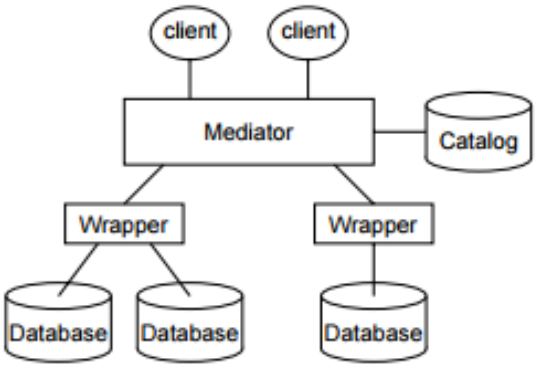

Рис. 5: Использование посредника для обработки разнородной информации [9].

В [9] представлены и такие важные аспекты, как алгоритмы оптимизации плана выполнения запросов и некоторые алгоритмы выполнения определенных операций, которые были использованы в рамках данной дипломной работы.

2. Архитектура исполнителя запросов

2.1. Функциональные требования к ядру исполнения запросов

Уточним требования к функциональным характеристикам исполнителя запросов:

- исполнитель запросов должен поддерживать возможность получения данных из разнородных источников (различных баз данных, документов, вебсайтов, и др.), причем ядро исполнения запросов должно быть легко масштабируемо относительно подключения различных источников;
- исполнитель запросов должен поддерживать возможность выполнения операций расширенной реляционной алгебры согласно плану выполнения запроса;
- необходима поддержка конвейерной обработки данных для уменьшения времени отклика и обеспечения равномерной загрузки хостов распределенной системы исполнения запросов.

2.2. Основные компоненты ядра исполнения запросов

Согласно приведенным выше функциональным требованиям, ядро исполнения запросов можно, в свою очередь, разделить на три основные функциональные компоненты.

Первой компонентой является планировщик, который получает план выполнения запроса от внешней компоненты системы (оптимизатора, или же непосредственно от языковых инструментов), на основе полученного плана определяет, какие компоненты ядра исполнителя необходимы для выполнения данного запроса, составляет для каждой из них задание, в которое входит подзапрос для данной конкретной компоненты, выделенный из общего плана выполнения запроса и некоторая мета-информация (например, адреса пересылки данных, адреса оповещения оповещений и др.)

Второй компонентой являются исполнители операций расширенной реляционной алгебры, организованные в древовидную структуру. Каждый исполнитель может поддерживать различные операции, алгоритмы их выполнения, используя различные технологии (например, средствами существующих баз данных, таких как PostgreSQL, AsterixDB, и др.) Следует отметить, что для добавления новых операций, достаточно лишь реализовать специальный интерфейс исполнителя, таким образом, достигается масштабируемость относительно поддержки новых операций и различных алгоритмов их выполнения.

Третьей компонентой являются адаптеры источников, которые, в свою очередь предоставляют интерфейс для извлечении данных из различных источников. Таким образом, достигается масштабируемость в отношении поддержки новых источников данных (то есть для извлечения данных из новой базы, или же типа документов, достаточно реализовать для него адаптер).

Следует отметить, что некоторые источники (использующие базы данных), имеют возможности для выполнения операций из плана выполнения, своими средствами, и могут так же реализовывать интерфейс исполнителя запросов.

2.3. Основные принципы функционирования исполнителя запросов

На рис. 6 представлена схема взаимодействия компонент ядра исполнения запросов.

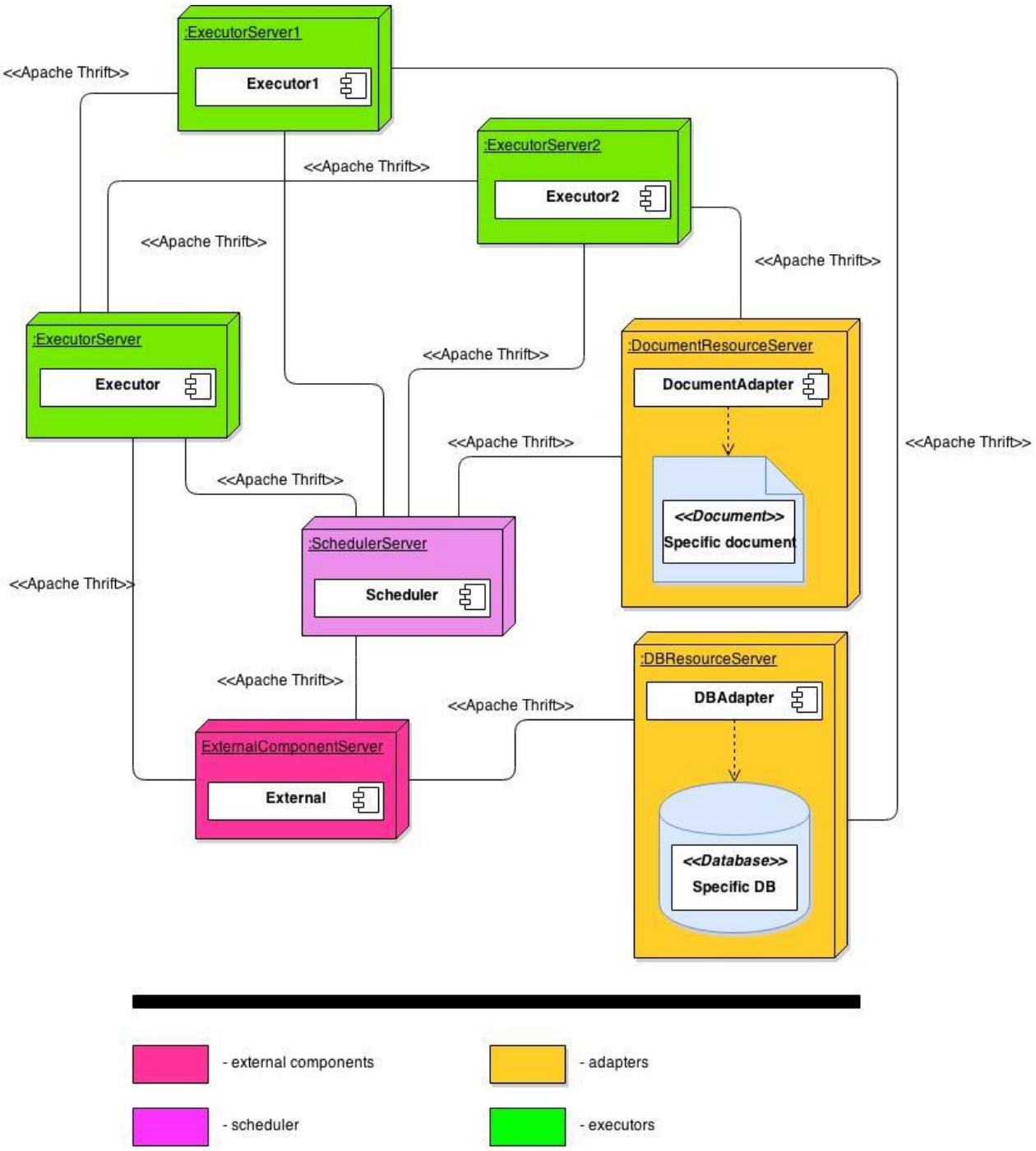

Рис. 6: Диаграмма развертывания ядра исполнения запросов.

Выполнение запроса начинается, когда планировщику поступает план выполнения запросов.

Планировщик формирует задания для каждого необходимого адаптера источника и исполнителя, в том числе и выделяет для них подзапросы и рассылает данные задания по хостам системы.

Далее, источники извлекают и обрабатывают информацию, согласно полученному заданию, и отправляют исполнителям.

Исполнители, как правило, организованы в древовидную структуру, в которой естественным образом, согласно плану выполнения общего запроса, организуется конвейерная обработка данных.

3. Поддержка конвейерной обработки данных

Для уменьшения времени отклика при выполнении запросов в реляционных базах данных может использоваться параллелизм, который в данном контексте можно разделить на два типа [14] :

- параллельное исполнение независимых операций,
- конвейерная обработка данных.

Первый вариант параллелизма в рамках разработанной архитектуры поддерживается с помощью выполнения независимых подзапросов на различных компонентах распределенного ядра исполнения запросов.

Конвейерная обработка данных осуществляется при выполнении последовательных операций как на одном из компонентов распределенной архитектуры ядра исполнения запросов, так и между отдельными исполнителями. Использование конвейерной обработки данных при выполнении операций расширенной реляционной алгебры [10] помимо уменьшения времени отклика может быть связано и с такими побочными эффектами, как увеличение объема памяти, необходимого для промежуточного хранения информации при выполнении тех или иных операций.

3.1. Организация конвейерной обработки данных при выполнении операции соединения

Рассмотрим поддерживаемые алгоритмы конвейерного выполнения операции соединения (*join*) [8]. Следует отметить, что существует несколько разновидностей алгоритмов для выполнения данной операции (*hash join, sort-merge join, nested-loop join*) [6]. С точки зрения организации конвейерной обработки при реализации данных алгоритмов различие заключается лишь в структуре, используемой для промежуточного хранения записей (*хэш-таблица, упорядоченный набор, неупорядоченный набор*) и методе

сопоставления записей между собой. Далее мы рассмотрим некоторые варианты конвейерной обработки данных при выполнении операции *hash join* (для *sort-merge join* и *nested-loop join* описанная далее процедура будет выглядеть аналогичным образом, с точностью, до упомянутых выше различий) [12, 15].

Первый вариант организации конвейерной обработки данных состоит из двух этапов (см. рис. 7):

- получение всех данных из одного источника и создание хэш-таблицы на их основе,
- последовательное получение и обработка записей со второго источника с использованием хэш-таблицы, построенной на первом этапе.

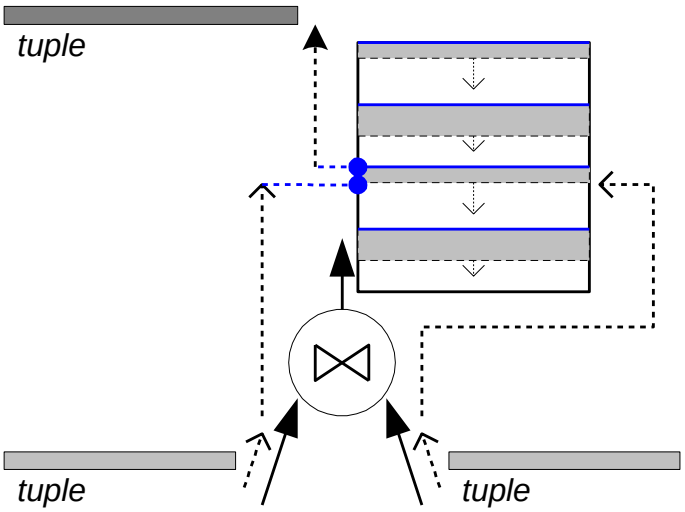

Рис. 7: Организация конвейерного исполнения операции hash join # 1 [6].

Второй вариант организации конвейерного выполнения операции *hash join* [5] возможно описать следующим образом. При получении записи из источника, происходит её обработка, согласно данным, имеющимся в хэш-таблице, соответствующей второму источнику, после чего, данная запись добавляется в хэш-таблицу, составляющуюся по информации из первого источника (см. рис. 8):

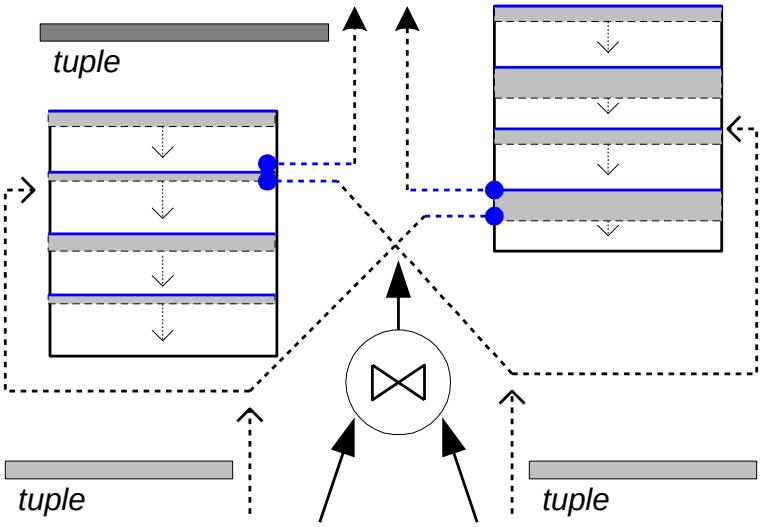

Рис. 8: Организация конвейерного исполнения операции hash join # 2 [6].

При данном способе выполнения операции соединения значительно сокращается время отклика и загрузка исполнителей происходит более равномерно (см. рис. 9). При этом возрастает и минимальный объем памяти, необходимый для выполнения данной операции, так как возникает потребность в построении второй хэш-таблицы.

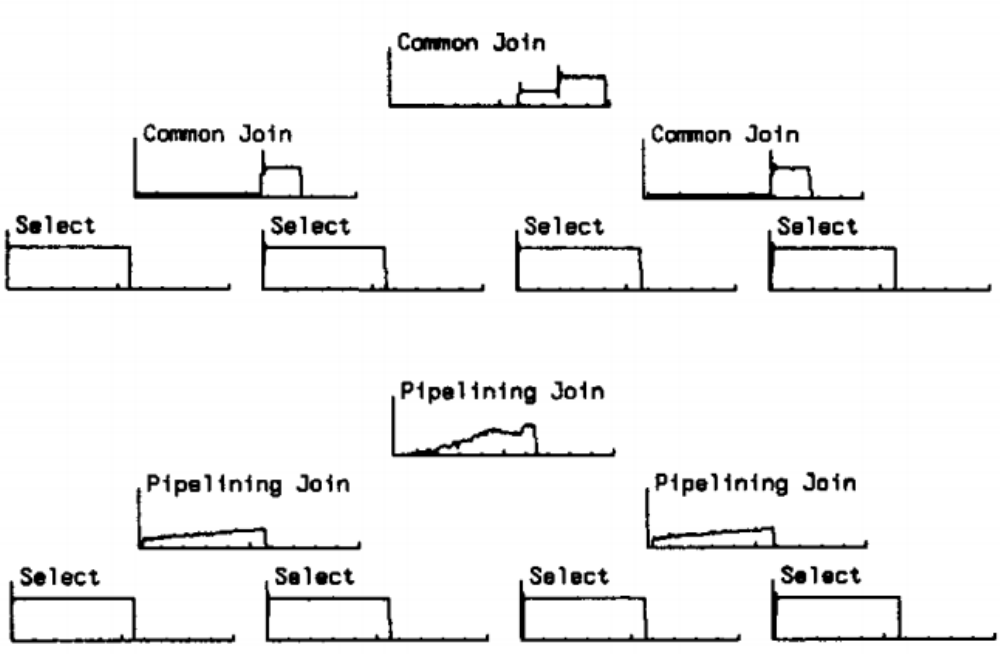

Рис. 9: Сравнение распределение загрузки вычислителя при выполнении операции соединения в случае конвейерной обработки (снизу) и в случае без конвейерной обработки (сверху) [14].

3.2. Организация конвейерной обработки данных при выполнении операции фильтрации

Конвейерная обработка данных при выполнении операции фильтрации может быть организована, как выполнение данной операции относительно последовательно поступающих фрагментов информации, что существенно уменьшает время отклика. Следует также отметить, что при выполнении данной операции помимо применения конвейерной обработки возможно организовать параллельную обработку данных на нескольких хостах системы для понижения нагрузки на отдельные узлы системы и ускорения обработки информации.

4. Интеграция с языковыми компонентами и оптимизатором

Для интеграции с языковыми компонентами системы обработки информации и оптимизатором, используется представление запросов в формате JSON (рис. 10), а так же, поддерживается передача запроса и данных по средствам бинарного транспортного протокола.

```
select * from Persons where age >= 22 and age <= 23
```


```
"1": {"str": "filterExpression"},
"2": {"i32": 28},
"9": {
    "lst": [
      "rec",
      1,
      {
        "1": {"str": ""},
        "2": {"i32": 15},
        "9": {
          "lst": [
            "rec",
            2,
            {
              "1": {"str": "age"},
              "2": {"i32": 23},
              "5": {"str": "age"}
            },
            {
              "1": {"str": ""},
              "2": {"i32": 22},
              "5": {"str": "30"}
            }
          ]
        }
      }
    ]
}
```


Рис. 10: Пример трансляции запроса в формат JSON.

4.1. Интерфейсы взаимодействия с внешними компонентами системы

Взаимодействие между планировщиком и языковыми компонентами (или оптимизатором) происходит по специально разработанному интерфейсу (рис. 11).

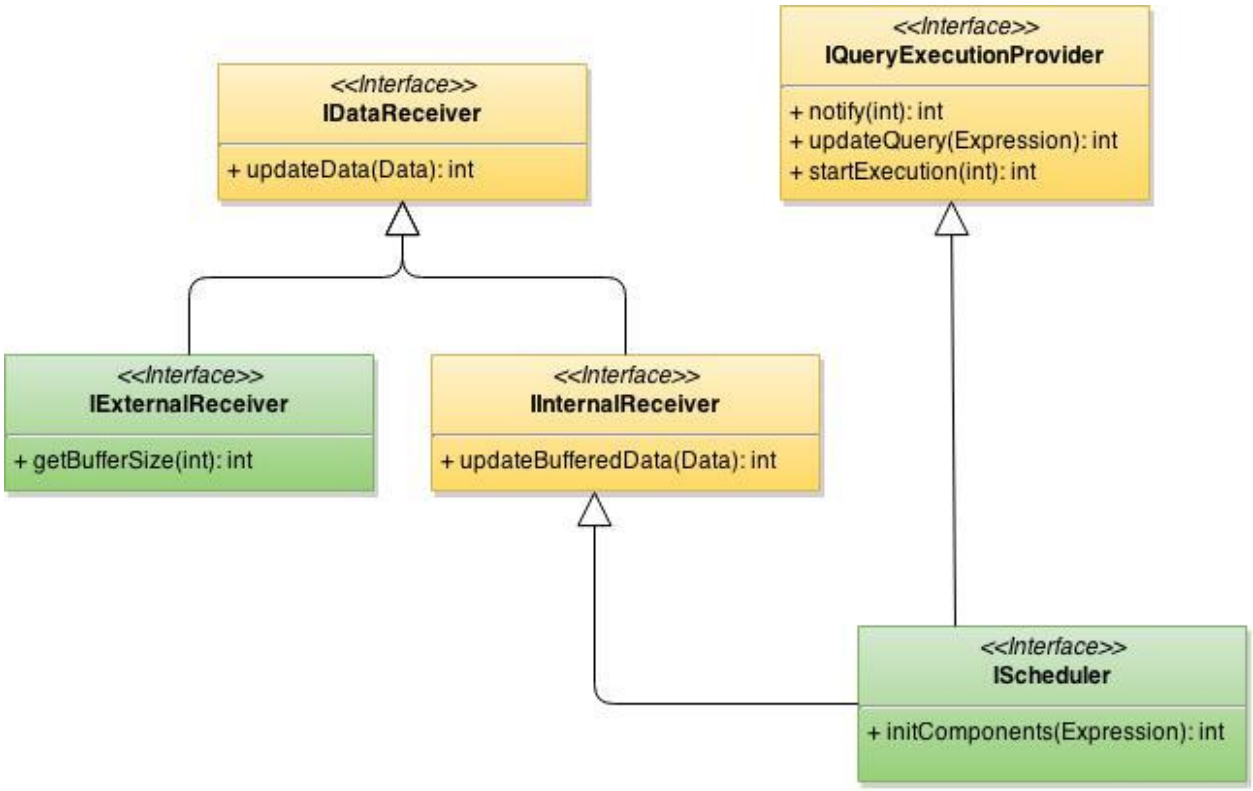

Рис. 11: Интерфейсы взаимодействия ядра исполнения и внешних компонент системы.

Согласно рис. 9, для организации взаимодействия между ядром исполнения запросов и внешними компонентами, необходимо реализовать интерфейс *IScheduler* в компоненте исполнителя запросов, отвечающей за получение и плана запросов от языковой компоненты, или же оптимизатора. Со стороны внешних относительно ядра компонент системы, в свою очередь, необходимо реализовать интерфейс *IExternalReceiver*, для возможности получения результата выполнения запроса.

Рассмотрим подробнее интерфейсы, представленные на рис.5:

- *IDataReceiver* - интерфейс компоненты, способной получать данные, содержащий метод:
  - *updateData(Data): int* – метод получения данных в унифицированном формате (*Data*) (см. приложение 1);
- *IExternalReceiver (extends IDataReceiver)* – интерфейс обработчика данных для внешних, по отношению к исполнителю запросов компонент системы, объявляющий метод:
  - *getBufferSize(int): int* – метод, возвращающий размер буфера для передачи данных внешней компоненте;
- *IInternalReceiver (extends IDataReceiver)* – интерфейс обработчика данных для внутренних компонент исполнителя запросов, объявляющий метод:
  - *updateBufferedData(int): int* – метод, необходимый для организации конвейерной обработки данных между исполнителями;
- *IQueryExecutionProvider* – общий интерфейс обработчиков запросов, содержащий методы:
  - *notify(int): int* – отправка оповещений (необходимо для синхронизации),
  - *updateQuery(Expression): int* – обновление конфигурации текущего запроса на выполнение (*Expression* – план выполнения запроса – см. приложение 2),
  - *startExecution(int): int* – начало обработки текущего запроса, согласно заданной конфигурации;
- *IScheduler (extends IInternalReceiver, IQueryExecutionProvider)* – интерфейс планировщика, объявляющий метод:

- *initComponents(Expression): int* – конфигурирование внутренних компонент ядра исполнения, согласно поступившей конфигурации запроса.

4.2. Унифицированный формат для внутреннего представления данных

Рассмотрим форму для унифицированного представления данных (см. рис.12).

Адаптеры источников приводят информацию к данному виду для последующего выполнения операций расширенной реляционной алгебры из [10] и передачи внешним компонентам системы обработки информации.

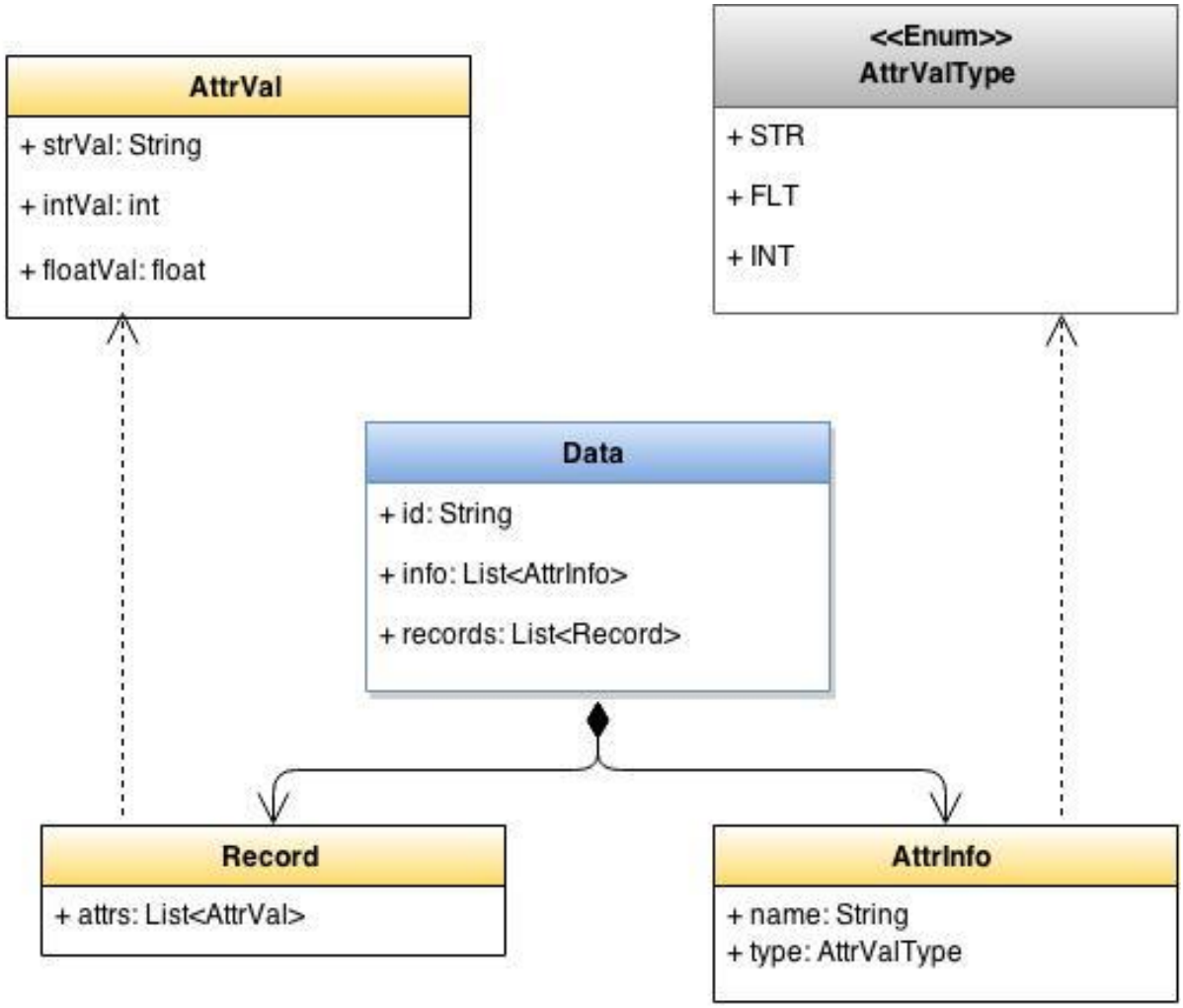

Рис. 12: Диаграмма классов для унифицированного формата представления данных.

Главным классом, определяющим формат данных, является класс *Data*, который содержит в себе описание мета-информации о структуре каждой записи (класс *AttrInfo*), а так же саму информацию, содержащуюся в записях (класс *Record*). В данной иерархии классов используется разделение объявления схемы представления информации и структуры хранения данных. Таким образом отпадает необходимость в дублировании мета-информации о строении элементов передаваемого отношения, что приводит к сокращению объемов памяти, необходимых для хранения информации и выполнения операций расширенной реляционной алгебры [10], а также обеспечивает ускорение передачи данных и снижение загруженности сети, и уменьшение времени отклика при исполнении запросов.

4.3. Формат для представления плана выполнения запроса

В процессе выполнения запроса пользователя внешние компоненты системы отправляют планировщику план выполнения запроса в определенном формате (см. рис. 13). Сами задания, которые отравляет планировщик исполнителям, содержат в себе подзапросы, также представленные в данном виде.

Класс *Expression* представляет собой описание узла в древовидной структуре плана выполнения запроса. Каждый узел представляет собой некоторое описание операции из расширенной реляционной алгебры [10], а также некоторую мета-информацию (например, алгоритм выполнения данной операции). Также с помощью иерархической структуры объектов данного класса производится описание выражений, используемых в качестве аргументов для операций из плана выполнения запросов.

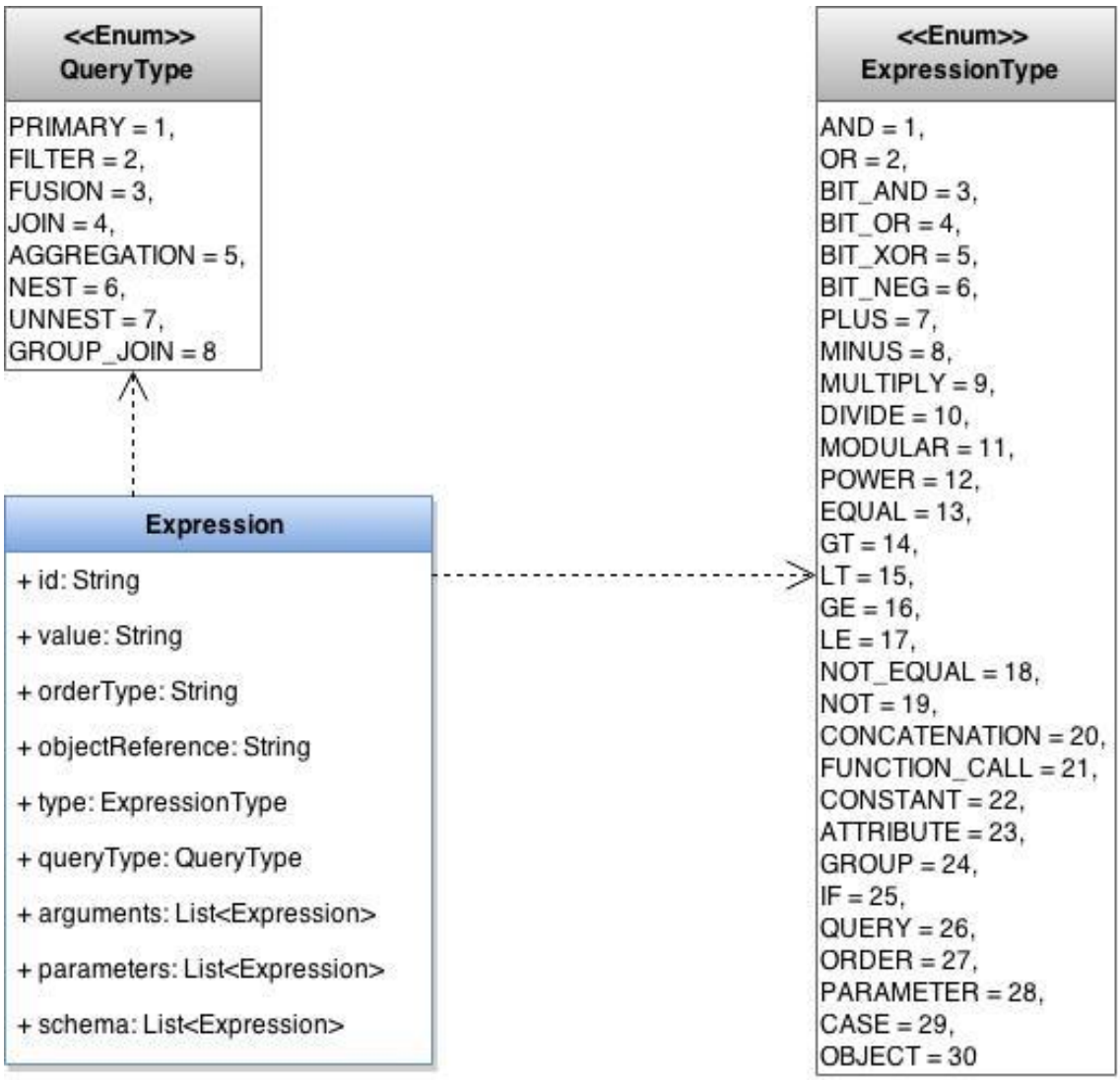

Рис. 13: Диаграмма классов для унифицированного представления плана выполнения запроса.

5. Интеграция исполнителей и адаптеров для источников

5.1. Интерфейсы исполнителей и адаптеров источников

Для реализации или интеграции существующих исполнителей и адаптеров источников необходимо привести существующий реализованный исполнитель запросов (например, см., [11]) к заданному общему интерфейсу исполнителей запросов (рис. 14).

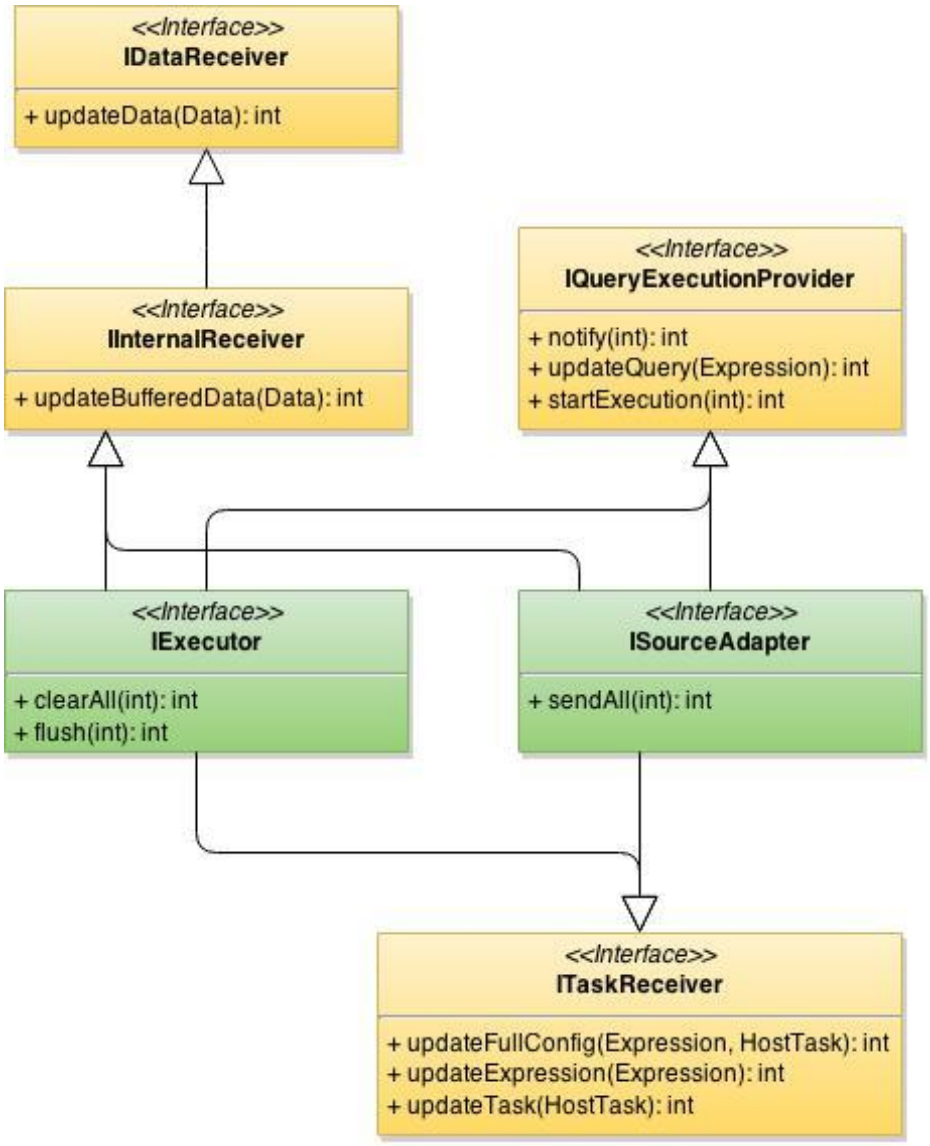

Рис. 14: Интерфейсы взаимодействия внутренних компонент ядра исполнения запросов.

Согласно рис.12, для подключения новых исполнителей и адаптеров источников к ядру исполнения запросов, необходимо реализовать интерфейсы IExecutor и ISourceAdapter, соответственно. Рассмотрим подробнее иерархию интерфейсов, представленную на рис. 6:

- *ITaskReceiver* – интерфейс, фиксирующий возможность передачи данной компоненте задания, включающего план выполнения запроса и некоторую мета-информацию, и содержащий следующие методы:
  - *updateFullConfig(Expression, HostTask): int* – получение плана выполнения запроса и мета-информации (*HostTask* – см. приложение 3),
  - *updateExpression(Expression): int* – обновление плана выполнения запросов,
  - *updateTask(HostTask): int* – обновление мета-информации;
- *ISourceAdapter(extends ITaskReceiver, IQueryExecutionProvider, IInternalReceiver)* – интерфейс адаптера источника, объявляющий метод:
  - *sendAll(int): int* – отправка всех имеющихся данных, согласно установленной мета-информации;
- *IExecutor(extends ITaskReceiver, IQueryExecutionProvider, IInternalReceiver)* – интерфейс исполнителя запросов, объявляющий следующие методы:
  - *clearAll(int): int* – очистка всех буферов,
  - *flush (int) :int* – обработка и пересылка всех накопленных данных, не дожидаясь заполнения буферов, с последующим их освобождением.

5.2. Представление мета-информации для конфигурирования исполнителей

Планировщик конфигурирует каждый адаптер источников и исполнитель с помощью мета-информации, заданной в определенном формате (см. рис. 15)

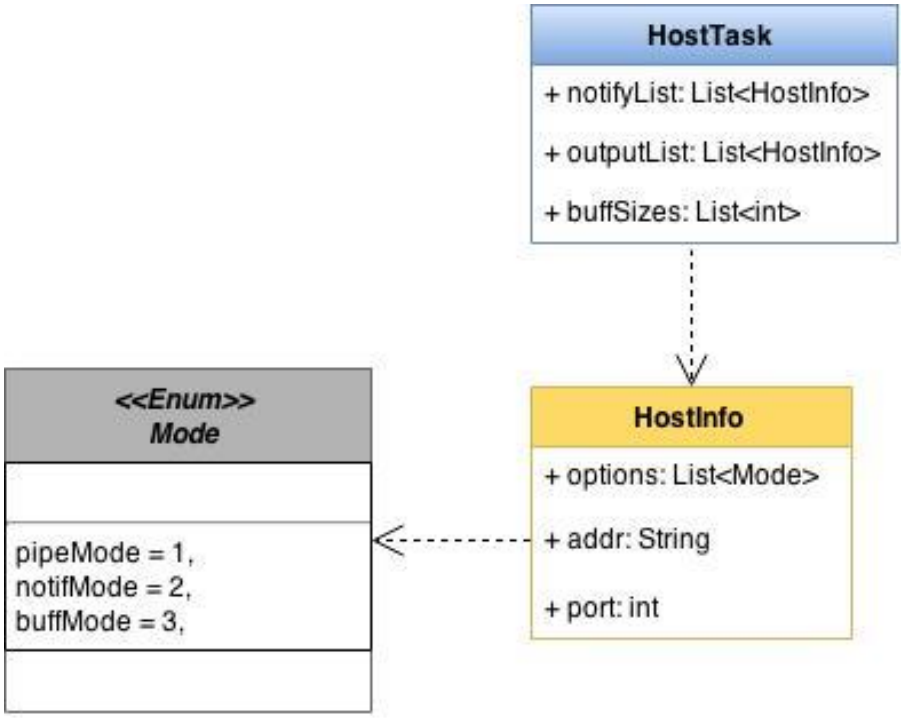

Рис. 15: Диаграмма классов для унифицированного представления мета-информации внутренних компонент ядра исполнения запросов.

Класс HostTask содержит информацию о хостах, которым следует передать данные, о хостах, которые следует оповестить (для синхронизации при выполнении запроса), и о размерах буферов для передачи информации. Данные о хостах, в свою очередь, представлены классом HostInfo, который содержит адреса, номера портов и устанавливаемые режимы для передачи информации или оповещений.

6. Апробация

6.1. Архитектура прототипа

Рассмотрим пример интеграции с языковыми компонентами и конкретными исполнителями на основе демонстрационного прототипа системы (рис. 16).

В качестве одного из адаптеров источников и исполнителя, был использован результат работы Николая Савельева по реализации операций расширенной реляционной алгебры на основе AsterixDB [13]. Данный компонент реализован с помощью *Java*, также как и языковая компонента (которая встроена в front end – на данной диаграмме (рис.14)). Остальные компоненты ядра исполнения запросов реализованы на *C++*, что демонстрирует еще одну особенность реализации ядра исполнения запросов - поддержку реализации отдельных компонентов системы с помощью различных технологий, на различных платформах.

6.2. Платформы и языки реализации

Таким образом, рассматриваемый демонстрационный прототип состоит из следующих частей: планировщик (*C++*), исполнитель, поддерживающий операцию join (*C++*), адаптер для текстовых файлов (*C++*), адаптер для AsterixDB (*Java*), front end с языковыми инструментами (*Java*). Для реализации коммуникации внутри ядра и с внешними компонентами, используется *Apache Thrift*. Хосты для развертывания прототипа сконфигурированы следующим образом:

- *JoinExecutorServer*:
  - аппаратное обеспечение: Intel Core i7 3630QM, 2.4 GHz, 8GB RAM,
  - программное обеспечение: x86_64 Ubuntu 14.04 LTS, ядро 3.13;

- *SchedulerServer*:
    - аппаратное обеспечение: Intel Core i7 4700MQ, 2.4 GHz, 8GB RAM,
    - программное обеспечение: x86_64 Ubuntu 14.04 LTS, ядро 3.13;

- *TextDataServer*:
    - аппаратное обеспечение: Intel Core i5 2430M, 2.4 GHz, 4GB RAM,
    - программное обеспечение: x86_64 Ubuntu 14.04 LTS, ядро 3.13;

- *AsterixDBServer*:
    - аппаратное обеспечение: Intel Core i5 3470, 3.2 GHz, 6GB RAM,
    - программное обеспечение: x86_64 Ubuntu 14.04 LTS, ядро 3.13;

- *FrontEndServer*:
    - аппаратное обеспечение: Intel Core i5 3470, 3.2 GHz, 6GB RAM,
    - программное обеспечение: x86_64 Ubuntu 14.04 LTS, ядро 3.13.

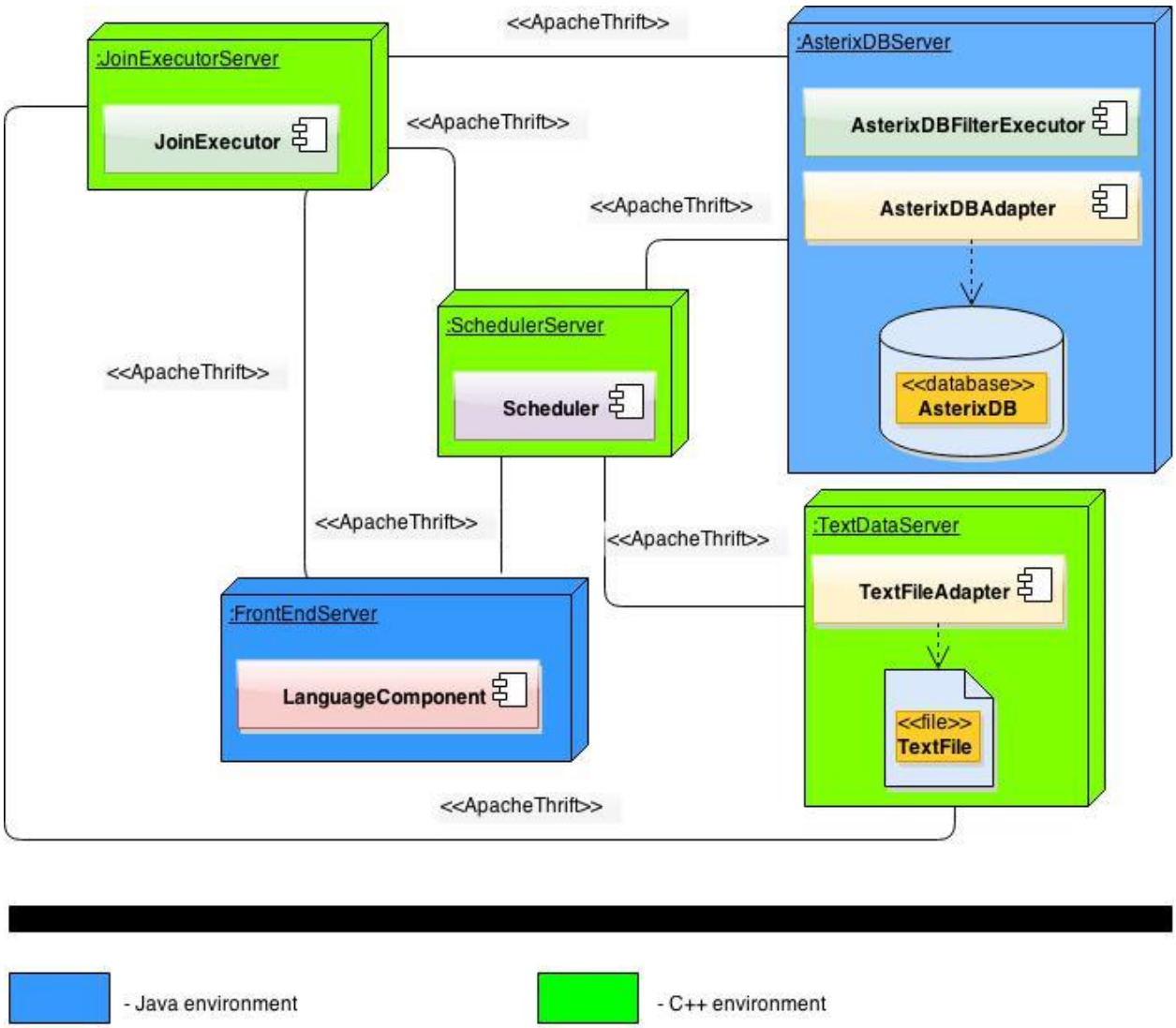

Рис. 16: Диаграмма развертывания прототипа.

6.3. Процесс обработки запросов

Рассмотрим подробнее работу данного прототипа.

Front end генерирует запрос. Здесь видно его представление в формате JSON и отправляет его планировщику (рис. 17).

Планировщик (*Scheduler*) генерирует задания для всех компонент (*Joiner*, адаптер для файла - *Document_adapter*, адаптер для Asterix_DB - *Asterix_DB_adapter*).

Адаптер файла извлекает данные из файла, отправляет их на исполнитель *join* (рис. 18).

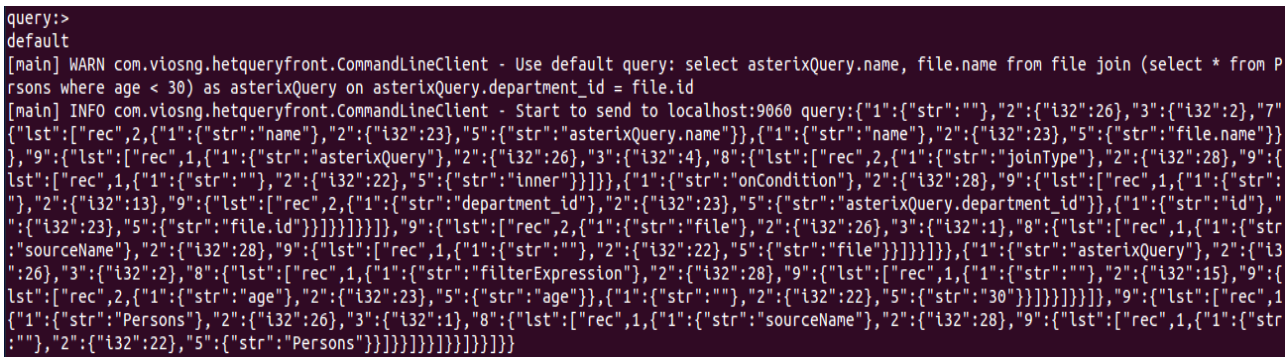

Рис. 17: Генерация и отправка запроса на планировщик.

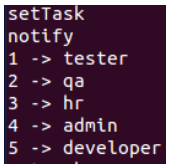

Рис. 18: Выполнение задания на адаптере файла.

Адаптер AsterixDB содержит в себе средства для выполнения запросов, и используется, как исполнитель операции *filter*, для чего получает соответствующий подзапрос вместе с заданием. Далее выполняется фильтрация с помощью средств AsterixDB, и отправка для выполнения *join* (рис. 19).

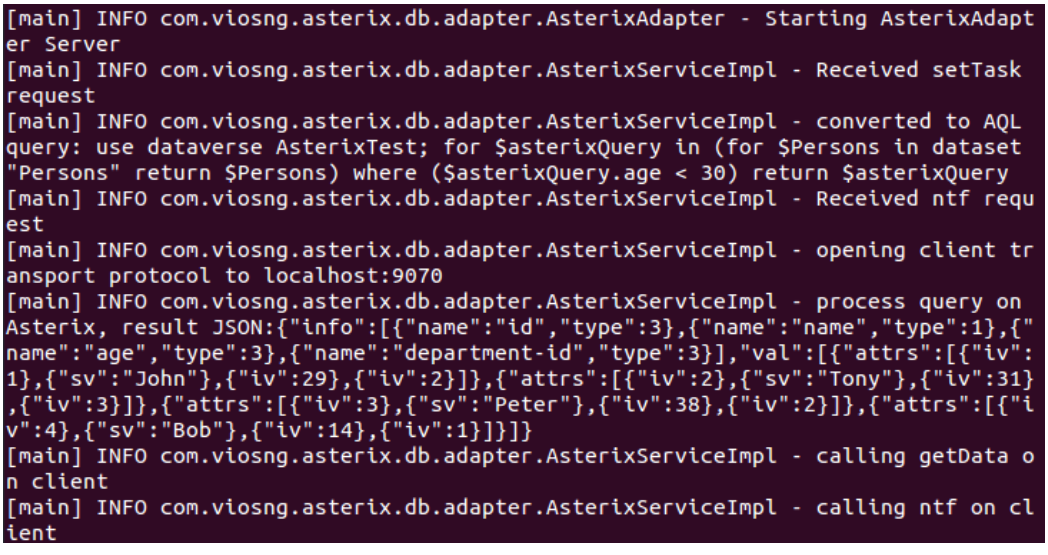

Рис. 19: Выполнение задания на адаптере AsterixDB.

Далее, выполняется *join*, и результат отправляется на *front end* (рис. 20).

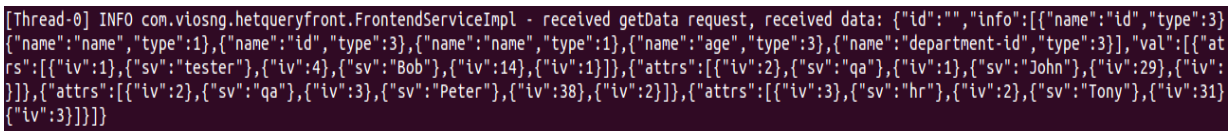

Рис. 20: Получение результата от исполнителя join.

Заключение

Результаты

В ходе выполнения данной работы были получены следующие результаты:

- разработана архитектура ядра исполнения запросов;
- обеспечена возможность интеграции с языковыми компонентами и оптимизатором:
  - разработаны интерфейсы взаимодействия с внешними компонентами системы,
  - разработан унифицированный формат для внутреннего представления данных,
  - разработан формат для представления плана выполнения запроса;
- обеспечена поддержка конвейерной обработки данных в рамках разработанной архитектуры;
- обеспечена возможность интеграции новых исполнителей и адаптеров источников:
  - разработаны интерфейсы для взаимодействия с исполнителями и адаптерами для источников информации,
  - разработан формат для конфигурирования компонент системы;
- произведена апробация (см. раздел апробация) разработанной архитектуры при взаимодействии с реализованным ранее исполнителем/источником данных на основе AsterixDB [13] и высокоуровневыми языковыми средствами для описания запросов. Вследствие чего был создан прототип системы с тестовым покрытием, что свидетельствует о выполнении функциональных требований к архитектуре ядра исполнения запросов (см. раздел архитектура исполнителя запросов);

- результаты выполнения данной дипломной работы представлены на ISSAT2015.

Следует отметить, что прототипы ядра исполнения запросов, реализованные на основе данной архитектуры, превосходят по производительности исполнителей, основанных на концепции *MapReduce* (например, на основе *Apache Hadoop* [17]), из-за возможности более гибкого управления стратегиями материализации. Также, из-за поддержки бинарной сериализации данных, будет наблюдаться и превосходство над системами, использующими формат JSON для внутреннего представления информации.

В дальнейшем, в рамках разработки рассматриваемой системы, планируются следующие шаги:

- интеграция с оптимизатором запросов,
- разработка адаптеров для поддержки открытых интерфейсов, таких как ODBC, JDBC, ADO, и др.,
- поиск и внедрение наиболее оптимальных стратегий для материализации данных [4],
- исследование алгоритмов исполнения нечетких запросов в рамках расширенной реляционной алгебры,
- исследование возможности использования методов машинного обучения для выполнения запросов.

Список литературы

[1] U. Dayal, S. Chaudhur, *An Overview of Data Warehousing and OLAP Technology*, ACM Sigmod Record, March 1997, pages 65-74.

[2] N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan, *Pipelining in Multi-Query Optimization*, In PODS, Journal of Computer and System Sciences, 2003, pages: 728–762.

[3] J. Gantz, D. Reinsel, *Big Data, Bigger Digital Shadow s, and Biggest Grow th in the Far East*, The Digital Universe in 2020, p 1 – 14, URL: https://www.emc.com/collateral/analyst-reports/idc-digital-universe-united-states.pdf.

[4] M. Grund, J. Krueger, M. Kleine, A. Zeier, H. Plattner, *Optimal Query Operator Materialization Strategy for Hybrid Databases*, DBKDA 2011, The Third International Conference on Advances in Databases, Knowledge, and Data Applications, 2011, pages: 169 - 174.

[5] P. Garcia, H. F. Korth, *Pipelined Hash-Join on Multithreaded Architectures*, DaMoN '07 Proceedings of the 3rd international workshop on Data management on new hardware, 2007.

[6] Z. Ives: *Query execution techniques*, CSE 544, Spring 2000, URL: http://courses.cs.washington.edu/courses/cse544/00sp/lectures/ppt/l5.ppt.

[7] Y. Ioannidis, *Query optimization*, ACM Comput. Surv. vol. 28, no. 1, 1996, pages. 121–123.

[8] J. Jayashree, C.Ranichandra, *Join algorithm for efficient query processing for large datasets*, AJCSIT, 2012, pages 31-35.

[9] D. Kossmann, *The State of the Art in Distributed Query Processing*, ACM Computing Surveys (CSUR) Surveys Homepage archive, Volume 32 Issue 4, Dec. 2000, Pages 422-469.

[10] B. Novikov, A. Vassilieva, A. Yarygina, *Querying Big Data*. In CompSysTech,12 Proceedings of the 13th International Conference of Computer Systems and Technologies, pages 1-10, 2012.

[11] K. Ong, Y. Papakonstantinou, R. Vernoux, *The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop*, NoSQL and NewSQL Databases, cs.DB, 19 Mar 2015 p 1-28.

[12] A. Shatdal, C. Kant, J. F. Naughton, *Cache Conscious Algorithms for Relational Query Processing*, VLDB '94 Proceedings of the 20th International Conference on Very Large, 1994, pages 510-521.

[13] N. Saveliev, *Implementation of generalized relational algebraic operations with AsterixDB BDMS*, New Trends in Database and Information Systems II Advances in Intelligent Systems and Computing Volume 312, 2015, pages: 323-332.

[14] A. Wilschut, P. Apers, *Pipelining in query execution*, PARBASE-90, 1990.

[15] H. Zeller, J. Gray, *An Adaptive Hash Join Algorithm for Multiuser Environments*, VLDB '90 Proceedings of the 16th International Conference on Very Large Data Bases, pages 186-197, 1990.

[16] *Embedding the Pentaho Reporting Engine*, URL: http://www.ambientelivre.com.br/downloads/doc_download/83-embedding-pentaho-reporting-engine.html.

[17] *Extract, Transform, and Load Big Data with Apache Hadoop*, URL: https://software.intel.com/sites/default/files/article/402274/etl-big-data-with-hadoop.pdf.

[18] *Pentaho Analysis Viewer User Guide*, URL: http://www.osbi.fr/wp-content/Pentaho-Analysis-Viewer-User-Guide.pdf.

[19] С.В.Скударнов, *Система DAMP: унифицированный доступ к гетерогенным данным в распределенных приложениях*, Системное

программирование. Том 4, вып. 1: Сб. статей / Под ред. А.Н.Терехова, Д.Ю.Булычева. - СПб.: Изд-во СПбГУ, 2009 г., с.:150-170.