Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ВЫСШАЯ ШКОЛА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ИНТЕЛЛЕКТУАЛЬНЫХ СИСТЕМ

Направление: 09.03.03 Прикладная информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА МЕХАНИЗМОВ ПОДДЕРЖКИ КОМБИНИРОВАННОГО ИСПОЛЬЗОВАНИЯ БАЗ ДАННЫХ В НЕСКОЛЬКИХ МОДЕЛЯХ ДЛЯ ФРЕЙМВОРКА DJANGO

Студент 4 курса

группы 11-503

«___»_____2019 г. Тимерханов Т.И.

Научный руководитель

старший преподаватель кафедры

программной инженерии

«___»_____2019 г. Абрамский М. М.

Директор Высшей школы ИТИС

«___»_____2019 г. Хасьянов А.Ф.

Казань – 2019 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На сегодняшний день наблюдается стремительный рост количества хранимой и обрабатываемой информации [1]. Эти данные являются разнородными, имеют различную структуру. В свою очередь, это привело к появлению многообразия хранилищ данных для более эффективного взаимодействия с разными видами информации.

На текущий момент наиболее распространенной является реляционная модель баз данных [2]. Реляционные БД предоставляют согласованность и целостность данных. В то же время они требуют заранее описанной структуры. Нереляционные решения, в свою очередь, являются более гибкими и хорошо подходят для частично структурированной и неструктурированной информации. Кроме того, с развитием технологии и увеличением объемов хранимых и обрабатываемых данных на первое место встаёт лёгкость горизонтального масштабирования и высокая скорость обработки запросов, которые также предоставляют базы данных NoSQL [3].

Каждый из подходов имеет собственные преимущества, которые проявляются при работе с различными видами информации. Стремление сочетать сильные стороны каждой БД привело к появлению гибридного подхода, сочетающего в себе SQL и NoSQL решения [4] [5] [6]. Основой такого подхода является не использование мультимодельных баз данных, а в использовании различных баз данных для разных задач в рамках приложения и обеспечения взаимодействия между этими базами.

В свою очередь, в силу распространенности реляционных хранилищ, большинство веб-фреймворков для взаимодействия с базой данных используют объектно-реляционное отображение (Object-Relational Mapping, ORM) [7]. Данное преобразование, как следует из названия, работает только с реляционными базами данных. Функциональность таких фреймворков во многом опираются на слой ORM, в связи с чем разработчик лишается возможности для совместного использования реляционных и нереляционных баз данных.

Вышеуказанные решения [4-6], предоставляют интерфейсы, которые работают (или симулируют эту работу) на уровне системы управления базами данных (СУБД). В данной работе рассмотрен подход, когда межмодельное взаимодействие переходит на уровень приложения. Пользователь-разработчик, самостоятельно определяет, к какому хранилищу принадлежат те или иные объекты отображения. При таком способе взаимодействия пользователь оперирует с интерфейсом, который похож на набор методов ORM, а перевод этих методов в нужный формат (в зависимости от места хранения объекта) и соблюдение целостности переходят на уровень разрабатываемого модуля.

Практическая значимость настоящей работы состоит в том, что будет предложен способ для комбинированного взаимодействия с базами данных многих моделей в рамках инфраструктуры популярного веб-фреймворка, что позволит использовать данное решение при разработке веб-приложений. Кроме того, единый интерфейс взаимодействия позволяет использовать нереляционные базы данных средством вызова методов привычного ORM, что позволит, например, использовать графовое хранилище, а взаимодействовать с ним такими же средствами, как с реляционным.

Целью данной работы является разработка инструмента для совместного взаимодействия приложения с базами данных различных моделей. Для реализации поставленной цели требуется решение следующих задач:

- Анализ средств для эффективного хранения информации о связях между объектами различных баз данных.
- Разработка модуля, который модифицирует средства стандартного объектно-реляционного отображения и позволяет совместно использовать несколько хранилищ для работы с данными.
- Предоставление унифицированного программного интерфейса приложения (API), который содержит методы для взаимодействия (добавление, редактирование, получение, удаление и пр.) с разными моделями.

- Сравнительный анализ производительности стандартных средств ORM и инструмента, который позволяет использовать несколько моделей данных.

1. ПОДХОДЫ К ОРГАНИЗАЦИИ ХРАНЕНИЯ ДАННЫХ

1.1. Модели баз данных

1.1.1. Реляционный подход

Реляционная модель была представлена Т.Коддом в 1970 [8]. В ее основе лежит модель, которая опирается на математическая модель, которая описывает отношения между множествами.

Реляционная база данных представляет собой множество связанных *отношений* (relations). Одной из привычных форм отношения может служить таблица состоящая из столбцов и строк. В формальной терминологии строка таблицы называется *кортежем* (tuple), заголовок столбца – *атрибутом* (attribute), тип данных для конкретного столбца описывается *доменом* (domain) возможных значений. Все записи таблицы имеют одинаковую, заранее определенную структуру. Для получения значения атрибута для объекта требуется указать название таблицы, сам атрибут и *первичный ключ*, который является однозначно определяет запись в таблице.

Для взаимодействия с реляционными базами данных используется язык SQL (Structured Query Language).

Большинство реляционных баз данных выполняют требования ACID (Atomicity, Consistency, Isolation, Durability – атомарность, непротиворечивость, изолированность, долговечность), которые обеспечивают наиболее надёжную и предсказуемую работу.

1.1.2. Нереляционный подход

Нереляционный (NoSQL) подход представляет собой объектно-ориентированные базы данных, которые предназначены для решения проблем, связанных с обработкой увеличивающихся объемов хранимых данных и разнообразной структурой этой информации. Необходимость в нереляционных базах особенно сильно проявляется в случаях, когда количество хранимой

информации намного превышает объемы, которые способны хранить реляционные БД, а также в случаях, когда структура этих данных такова, что эффективное хранение в реляционной модели является проблематичным или даже невозможным [5]. Базы данных NoSQL спроектированы на распределенной модели, чтобы гарантировать принципы BASE (Basically Available, Soft-state, Eventually consistent) [9]. Существует четыре основных модели нереляционных баз данных [10] [11]:

- Типа **Ключ/Значение**: Данные хранятся как совокупность пар «ключ-значение», в которых ключ служит уникальным идентификатором для доступа к значению. Как ключи, так и значения могут представлять собой простые и составные объекты. Базы данных с использованием пар «ключ-значение» поддерживают высокую разделяемость и обеспечивают беспрецедентное горизонтальное масштабирование, недостижимое при использовании других типов баз данных. [12]. Пример: DynamoDB, Riak, Redis (in-memory).
- **Семейство столбцов**: В таких системах данные хранятся в виде разреженной матрицы, строки и столбцы которой используются как ключи. Пример: Cassandra.
- **Графовые**: Основной структурой для хранения данных является граф. Узлы (nodes) хранят информацию об объектах и связаны друг с другом отношениями, которые также могут хранить в себе данные. Графовые базы данных обеспечивают высокую производительность при работе с данными, которые имеют большое количество связей [13]. Кроме того, они так же более производительных в задачах поиска, связанными с теорией графов (Рекурсивный поиск, поиск кратчайшего пути) [14]. Пример: Neo4j.
- **Документоориентированные**: В их основе лежат документные хранилища, имеющие иерархическую структуру. Такая древовидная структура начинается с корня и может содержать множество поддеревьев. Листовые узлы содержат данные, которые заносятся в индексы, что

позволяет эффективно искать эти данные даже при достаточно сложной структуре. Пример: MongoDB, CouchDB.

1.2. Сравнение реляционных и нереляционных баз данных

Нереляционные базы данных были разработаны для того, чтобы преодолеть недостатки и ограничения реляционного подхода. Сильные и слабые стороны каждого из подходов необходимо рассматривать в контексте масштабируемости и требований, которые предоставляют сами данные.

1.2.1. Масштабируемость

Увеличение объемов хранимых данных для реляционных систем осуществляется путем замены диска или сервера на более производительные, т.е. добавлением аппаратного обеспечения. Такой способ называется вертикальным масштабированием. Вертикальное масштабирование является очень дорогим и непрактичным в силу ограничений аппаратных средств [15]. Нереляционные базы данных основаны на распределенной архитектуре, что делает возможным разделение (sharding) базы между несколькими серверами. Таким образом, расширение достигается за счет добавления в кластер базы данных недорогих серверов. Такой подход называется горизонтальным масштабированием. Горизонтальное масштабирование повышает производительность системы при минимальных затратах, обеспечивая быстрое расширение объемов рабочих данных и устраняя единую точку отказа, существующую в реляционных базах данных.

1.2.2. Зависимость от схемы данных

Реляционные базы данных зависят от схемы данных. Данные не могут быть сохранены без определения схемы. В современных реалиях, где

увеличивается объем неструктурированной и частично структурированной информации, развивается область больших данных [16], невозможно заранее определить схему. Нереляционная модель, в свою очередь, имеет динамическую схему, которую не нужно определять заранее. Таким образом, нереляционные БД подходят для хранения как структурированной, так и неструктурированной информации.

1.2.3. Транзакции

Как уже было сказано, реляционные БД соответствуют требованиям ACID. Это позволяет уменьшить вероятность непредсказуемого поведения системы и обеспечить целостность и сохранность базы данных. Для достижения таких результатов требуется жестко определять, как именно транзакции взаимодействуют с базой данных. Это существенно отличается от требований BASE, выполнение которых обеспечивают. NoSQL базы. Эти требования является гораздо более мягкими, т.к. нереляционные базы на первое место ставят гибкость и скорость работы, а не стопроцентную целостность данных.

1.3. Мультимодельные базы данных и Polyglot Persistence

В отличие от традиционных СУБД, которые организованы вокруг единственной модели данных, которая определяет как организована, как хранится и обрабатывается эта информация, мультимодельные базы данных предназначены для поддержки нескольких моделей в рамках одного сервера.

Основное различие между доступными мультимодельными базами данных связано с их архитектурой. Мультимодельные базы данных могут поддерживать различные модели либо в движке, либо через разные слои поверх движка. Некоторые продукты могут предоставить движок, который поддерживает документы и графы, в то время как другие предоставляют слои поверх хранилища ключей [17].

Другим подходом для совместного использования многих моделей данных является использование нескольких баз данных (Polyglot persistence, Fowler, M.) [4]. Его суть состоит в использовании определенной базы для определенных данных и задач. Например, на рис. 1 можно видеть пример использования реляционной базу данных для хранения структурированных табличных данных, для которых важна их целостность и доступность (информация о платежах); документной - для неструктурированных (каталожная информация), структура которых может меняться со временем, а графовой для сильно связанных ссылочных (иерархических) данных (рекомендательные системы).

Рисунок 1 – Подход Polyglot persistence для приложения интернет-магазина

Такая стратегия имеет 2 основных недостатка – увеличение оперативной сложности взаимодействия с несколькими СУБД и отсутствие согласованности данных в отдельных хранилищах. Но именно на решение этих проблем и направлен разработанный инструмент. Кроме того, на данный наблюдается недостаток инструментов для работы с мультимодельными базами и использования их в разработке приложений. В свою очередь, базы данных NoSQL являются популярными и существует множество драйверов, отображений и других инструментов для использования их в разработке.

2. ОБЩАЯ АРХИТЕКТУРА

2.1. Используемые технологии

Для реализации инструмента был выбран язык Python. Основой для мультимодельного взаимодействия является Django ORM, который является ключевой частью веб-фреймворка Django[1]. Объектно-реляционное отображение, которое предоставляет Django является мощным и в то же время простым инструментом для взаимодействия со слоем базы данных (реляционной). Кроме того, были использованы и другие отображения:

- Neomodel[2] – Object Graph Mapping для взаимодействия с графовой базой данных neo4j.
- Djongo[3] – Object Document Mapping для взаимодействия с документоориентированной базой данных MongoDB.

В целом реализация не зависит от конкретных технологий, данная концепция может быть реализована и на основе других фреймворков (Spring + Hibernate, Ruby On Rails + Active Record). Наличие реализаций OGM и ODM также не является необходимым, достаточно наличие драйвера для доступа к БД.

2.2. Структура взаимодействия

Стандартный Django ORM для работы с данными использует две основные структуры [18]:

---

[1] https://www.djangoproject.com/ – The web framework for perfectionists with deadlines.

[2] https://neomodel.readthedocs.io/en/latest/ – An Object Graph Mapper (OGM) for the neo4j graph database

[3] https://nesdis.github.io/djongo/ – Django MongoDB connector

- *Model* - отображает информацию о данных. Содержат поля и поведение для этих данных. Django предоставляет автоматически созданное API для работы с моделями;
- *QuerySet* - необходим для “ленивого” поиска набора объектов. Он может быть создан, отфильтрован, ограничен и использован фактически без выполнения запросов к базе данных. База данных не будет затронута, пока не будет спровоцировано выполнение QuerySet.

В свою очередь, взаимодействие с нереляционными базами данных требует собственного объектного отображения (Object-Graph Mapping, Object-Document Mapping) в зависимости от используемой модели, каждое из которых имеет собственное API. В связи с этим, необходим механизм, который позволяет оперировать сущностями одной базы данных средствами одного конкретного отображения, а при запросе связанных объектов подключать и другие (внешние) преобразования. Это делегирование должно осуществляться на уровне специального слоя взаимодействия между базами данных. Кроме того, данный слой должен контролировать работу со связями и их целостностью, а также позволять фильтровать объекты по внешним связям. Схема представлена на рис. 2.

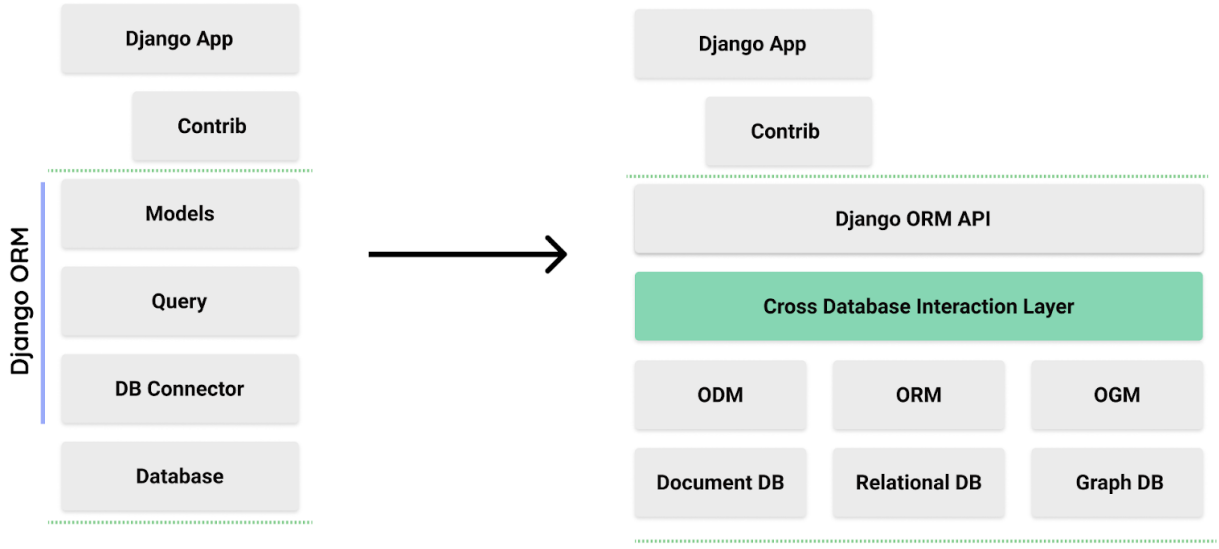

Рисунок 2 – Слой для взаимодействия с базами данных многих моделей

2.2.1. Инициализация классов

Для использования внешних связей в классе-модели (в терминах Django ORM) должны быть определены поля связей, которые являются экзмемплярами необходимых классов (*RDRelationship*, если связанные объекты хранятся в реляционной БД, *GraphRelationship*, – в графовой и *DocumentRelationship*, – в документоориентированной).

```
author = RDRelationship('authors.models.Author',
                        id_strategy=django_cdbms.INNER,
                        related=True)
```

Каждый класс связи на вход получает:

- Класс связанной модели или строку, содержащую его название. Название необходимо для «ленивой» загрузки этих классов, что позволяет избежать зацикленных импортов.
- Стратегию хранения информации о внешних связях (`id_strategy`). Доступны два способа – внутренний (`django_cdbms.INNER`) и внешний (`django_cdbms.OUTER`). Их подробное описание – в следующем разделе.
- Указание на необходимость обращения к объектам данного класса со стороны класса целевой модели.

Для корректной работы каждый класс-модель, который имеет внешние связи, также должен наследоваться от класса `CrossDBModel`, а также использовать специальный метакласс `CrossDBModelMeta`. Экземплярами этого метакласса являются классы-наследники `CrossDBModel`. Каждый из классов-наследников класса `CrossDBModel` содержит список всех межмодельных связей. Их определение происходит на этапе загрузки этих классов в память [19]. При создании экземпляров метакласса (вызов `CrossDBModelMeta.__new__()`) динамически находятся все межмодельные связи и список всех таких связей `__all_cross_relationships__` становится атрибутом класса. При инициализации непосредственно объектов классов-наследников `CrossDBModel` вызов этих полей-связей заменяется на вызов соответствующего менеджера,

который на вход получает сам объект и в дальнейшем переводит запросы к Django ORM в запросы к необходимому отображению.

```
@classmethod
 def defined_cross_relationships(cls, rels=False):
    props = {}
    for baseclass in reversed(cls.__mro__):
        props.update(dict(
               (name, property) for name, property in
                   vars(baseclass).items()
                if rels and isinstance(property,
                                       CrossDBRelationship)
            ))
    return props
```

Кроме того, для внешнего способа хранения информации о связях предусмотрена автоматическая генерация полей-связей у классов связанных объектов. Например, в классе Parent было определено поле связи с классом Child и `id_strategy=cdbms.OUTER` и `related=True`. В таком случае нет необходимости явно определять поле связи в классе Child. Оно будет сгенерировано автоматически на этапе определения всех классов-наследников `CrossDBModel`. Эта функциональность также реализована в метаклассе `CrossDBModelMeta`. На этапе инициализации атрибут `__all_cross_relationships__` связанного класса дополняется связью на родительский класс.

2.2.2. Классы-менеджеры

Основная роль менеджеров состоит в оперировании с внешними связями и переводе cross-database запросов в формат конечной базы с учетом способа хранения данных связанных объектов. Экземпляры классов менеджеров хранят:

- Родительский объект (source class) – объект, у которого вызывается поле межмодельной связи.
- Класс родительского объекта.

- Класс связанных объектов (target class) – необходим для обращения к объектам этого класса средствами объектных отображений.
- Название поля.
- Название поля для хранения списка идентификаторов (опционально) – используется для получения связанных объектов, если идентификаторы хранятся внутри родительского объекта.

Данный набор параметров позволяет осуществлять полный набор действий со связанными объектами. Схема организации процесса такого взаимодействия представлена на рис. 3.

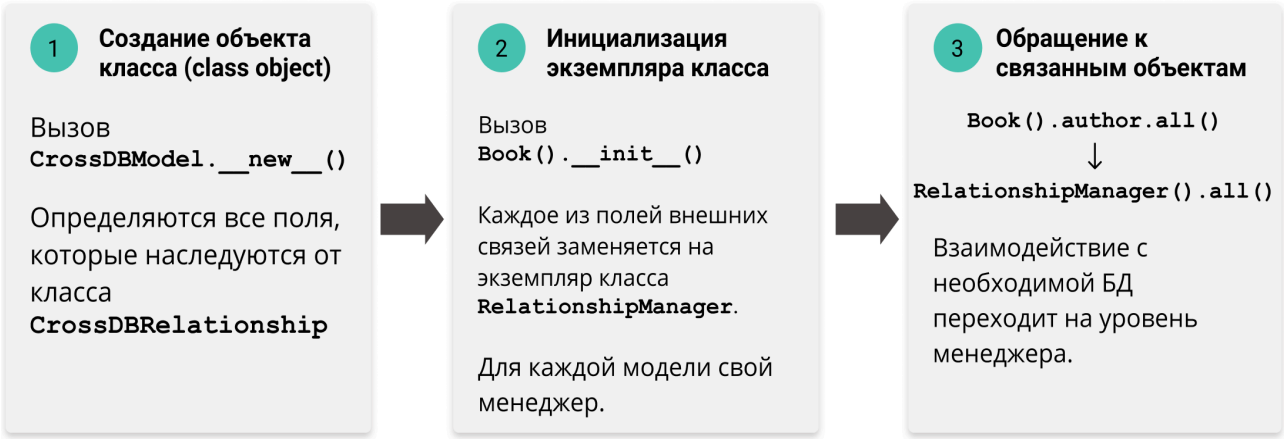

Рисунок 3 – Схема подготовки классов-моделей

2.3. Стратегии хранения данных о внешних связях

Для работы с объектами, которые хранятся в хранилищах разных моделей, необходимо хранить информацию о таких связях. В данной версии модуля предусмотрено два подхода к хранению межмодельных связей.

2.3.1. Хранение внутри объекта БД (INNER)

Набор идентификаторов хранится непосредственно в поле сущности базы данных (рис. 4).

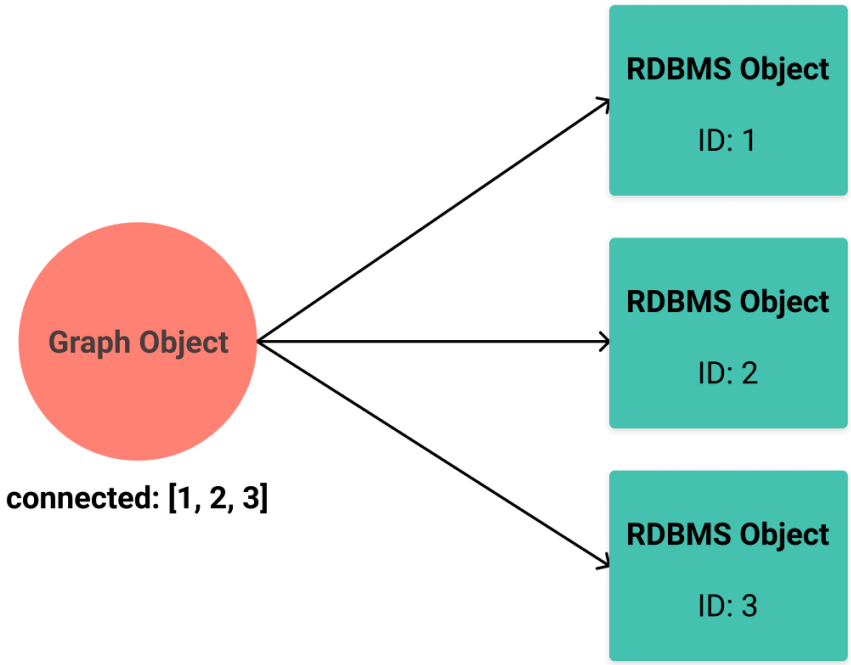

Рисунок 4 – Хранение идентификаторов внутри объекта

Такой подход позволяет получить связанные объекты за 2 запроса:

1. Запрос к родительскому объекту.
2. Запрос к внешней по отношению к родителю базе для получения связанных объектов по их идентификаторам.

Для корректной работы в обоих направлениях такой подход требует контроля значений идентификаторов при добавлении или изменении связей. Например, необходимо получить сущности класса Parent и связанные с ними сущности класса Child, которые содержатся в хранилищах разных моделей. Для дальнейшего получения объектов в обоих направлениях (parent.children и child.parents), необходимо выполнить две операции:

```
parent.children = ids(children)        (1)
FOR child in children:
     child.parents += parent.id.       (2)
```

где **ids()** - функция для получения массива идентификаторов из набора объектов.

Если заранее известно, что связь требует получения объектов только в одну сторону (например, требуется получить все объекты типа Child для

объектов типа Parent, но не наоборот), то, достаточно взаимодействия с массивом идентификаторов дочерних сущностей.

2.3.2. Хранение информации о связях в отдельном хранилище (OUTER)

Данный подход предполагает хранение информации о внешних связях в отдельном сервисном хранилище. Это позволяет взаимодействовать с информацией о межмодельных связях в рамках единой сервисной СУБД. В текущей версии модуля реализовано хранение такой информации в графовом и реляционных хранилищах.

Графовая база данных хранит узлы, которые содержат в себе поле уникального идентификатора из собственного хранилища. Узлы объединены в метки, которые соответствуют названию класса. Как уже было сказано в главе 1, графовые СУБД хорошо подходят для оперирования со связанными данными, а также не требуют заранее спроектированной структуры, что позволяет добавлять связи между разными сущностями по мере их появления.

3. ОПЕРАЦИИ НАД ОБЪЕКТАМИ РАЗНЫХ МОДЕЛЕЙ

3.1. Получение связанных объектов

За получение связанных объектов отвечают методы `all(), filter()` которые реализованы в классах-менеджерах. Пример использования данных методов:

```
# Получение объекта класса Parent из реляционной БД
parent = Parent.objects.get(uid=1)

# Получение всех связанных объектов
children = parent.child.all()

# Получение связанных объектов, имя которых содержит "Ива"
children = parent.child.filter(name__icontains='Ива')
```

Для получения объектов из смежной модели необходим список идентификаторов, по которой они будут найдены в целевой БД. При внутреннем способе хранения идентификаторов достаточно обратиться к полю родительской сущности. При хранении данных о связях во внешнем хранилище необходимо сделать запрос на получение связанных объектов в рамках этого хранилища. Ниже представлен пример подобного запроса для базы данных Neo4J на языке Cypher:

```
MATCH (p:Parent)-[:HAS_CHILDREN]->(c:Child)
WHERE p.uid='123abc'
RETURN c.uid
```

В результате, возвращается список идентификаторов. Фильтрация (поиск по заданным условиям) связанных объектов осуществляется на этапе получения их из собственной БД. При этом комбинируются (логический оператор &) два условия:

1. Идентификатор запрашиваемого объекта находится в массиве, полученного из сервисного хранилища.
2. Выполняется дополнительный набор условий для фильтрации этих объектов.

Данный запрос формируется стандартными средствами используемых объектных отображений.

3.2. Добавление межмодельных связей

Создание межмодельных связей происходит либо при вызове метода класса-менеджера `connect()`, либо при присваивании полю связи нового значения (вызов `field.setter()`). Второй способ необходим для обеспечения обратной совместимости с Django ORM API. Также, метод `connect()` позволяет добавлять связи к уже существующим. Например:

```
# Получение объектов, которые необходимо присоединить
children = Child.nodes.filter(uid__in=[1,2,3])
# Получение родительского объекта, к которому необходимо присоединить дочерние
parent = Parent.objects.get(uid='abc')
# Вызов метода connect()
parent.child.connect(children)
# Присвоение значения полю
parent.child = children
```

При хранении списка идентификаторов метод класса-менеджера берет идентификаторы объектов, поданных в качестве параметра, и добавляет их в массив. Если эти объекты присваиваются полю связи, то массив сначала обнуляется. Если при определении связи передан параметр `related=True`, то к каждому массиву из переданных объектов также добавляется идентификатор родительского объекта.

При использовании внешнего хранилища необходимо добавить информацию об этих связях. Для этого менеджер берет идентификаторы объекта источника и объектов, которые будут присоединены. Если узел с одним из этих идентификаторов существует, то возвращается уже созданный. Пример сгенерированного Cypher-запроса выглядит следующим образом:

```
MATCH (p:Parent)
WHERE p.uid='62sd2d2c-356s-4nh4-bsa5-9s4cd56v5bdd'
UNWIND  [
     {uid: "3509429e-0c92-42d7-a7df-420076360a90"},
     {uid: "fd9354d2-307f-4596-8575-04c16b69b839"},
     {uid: "cef20eec-2c3b-439b-b725-35a17ccd03ee"},
     {uid: "81c4ad2c-3a69-4d89-ba85-9b4cdcc906eb"},
] as row
MERGE (c:Child {uid: row.uid})
MERGE (p)-[:IS_CONNECTED]->(c)
```

3.3. Удаление связей

Для удаления межмодельных связей используются методы `remove()` и `clear()`.

```
parent = Parent.objects.get(uid=1)
child = Child.nodes.get(uid='abc')
parent.children.remove(child)
```

Метод `remove()` необходим для удаления единичного объекта или набора объектов. Если объекты хранят идентификаторы связанных сущностей внутри себя, то необходимо удалить каждый полученный методом идентификатор из этого множества у родительской сущности и, если классу связи передан параметр related=True, из списка связанных идентификаторов дочерних объектов удалить идентификатор родительского.

Если используется сервисное хранилище, то необходимо найти все дочерние узлы и удалить связь с родительским. Пример запроса к сервисному хранилищу:

```
MATCH (p:Parent)
WHERE p.uid='62sd2d2c-356s-4nh4-bsa5-9s4cd56v5bdd'
MATCH (p)-[rel:IS_CONNECTED]->(c:Child)
WHERE c.uid IN
[
     {uid: "3509429e-0c92-42d7-a7df-420076360a90"},
     {uid: "fd9354d2-307f-4596-8575-04c16b69b839"},
     {uid: "cef20eec-2c3b-439b-b725-35a17ccd03ee"},
     {uid: "81c4ad2c-3a69-4d89-ba85-9b4cdcc906eb"},
]
DELETE rel
```

Метод clear() удаляет все связи между заданными объектами. Что эквивалентно очищению каждого множества идентификаторов или удалению всех связей между родительским узлом и всеми связанными объектами, которые имеют метку связанного класса.

4. СРАВННИТЕЛЬНЫЙ АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ КЛАССИЧЕСКОГО И КОМБИНИРОВАННОГО ПОДХОДОВ

В рамках данного тестирования не ставится цель провести сравнение различных систем управления базами данных. Необходимо показать, что использование разных моделей для разных видов данных со связями между разными моделями в рамках разработанного модуля может приводить к увеличению производительности, а в некоторых случаях предоставлять функциональность, которая не могла бы быть реализована в случае использования единственного хранилища. В силу того, что проводилось сравнение баз данных различных моделей, были разработано N различных сценариев для эталонного тестирования.

Время выполнения операции является базовым критерием для сравнения производительности веб-приложений. Чем меньше времени занимает обработка запроса, который пришел от пользователя, тем более производительной является система. На основе разработанных сценариев были проведены эксперименты для получения временных оценок их выполнения. Каждый сценарий запускался 100 раз для того, чтобы получить среднее время выполнения. Лучший и худший результаты не учитывались, чтобы исключить влияние системы кэширования или задержек в работе операционной системы. Замер времени осуществлялся в миллисекундах (мс.).

Для каждой базы данных был использованы драйверы последних версий, рекомендованных производителями.

- MongoDB v4.0.10, PyMongo v3.8.0
- Neo4j 3.5.6, neo4j-driver v1.7.1
- PostgreSQL v11.3, psycopg2 v2.8.3

4.1. Запросы в рамках одной модели

Получение данных.

Внутри приложения для тестирования содержались идентификаторы объектов, которые было необходимо получить из базы данных. Структура не содержала большого количества связей или других особенностей, присущих той или иной модели данных. Каждый запрос состоял в получении объектов по списку идентификаторов. В разработанном модуле данные хранились в документоориентированной базе.

Таблица 1 – Результаты выполнения запросов на получение данных, в миллисекундах

Объекты	PostgreSQL	MongoDB	Neo4J	Прототип
1000	200	320	1160	333
10000	510	380	1310	328
100000	1100	600	1621	698
1000000	15100	4800	5401	4736

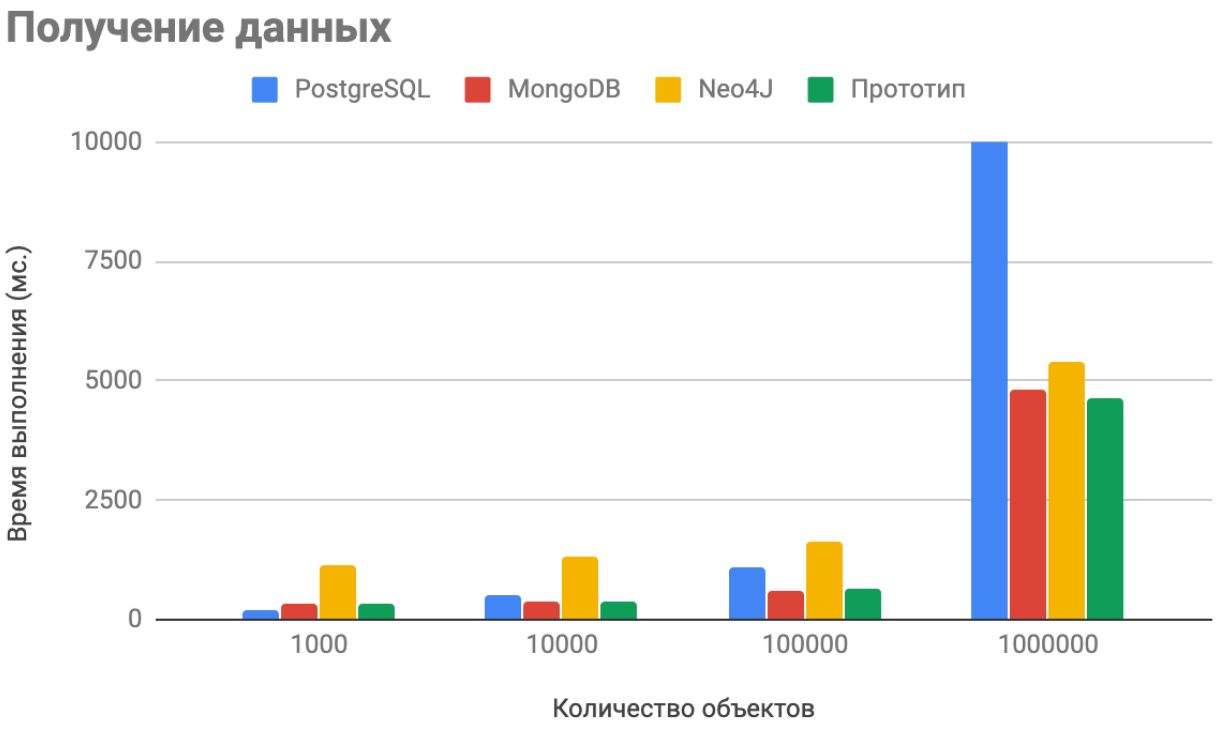

Рисунок 5 – Результаты выполнения запросов на получение данных

Внесение данных.

Загрузка объектов отдельными запросами с ожиданием успешного ответа от базы данных. Данные сгенерированы заранее и хранятся в файле JSON. При использовании разработанного модуля данные вносились в документоориентированную базу.

Таблица 2 – Результаты выполнения запросов на получение данных, в секундах

Объекты	PostgreSQL	MongoDB	Neo4J	Прототип
1000	0,7	2,5	14	3,5
10000	6	7,2	18,4	7,2
100000	74	69,5	101	71,5

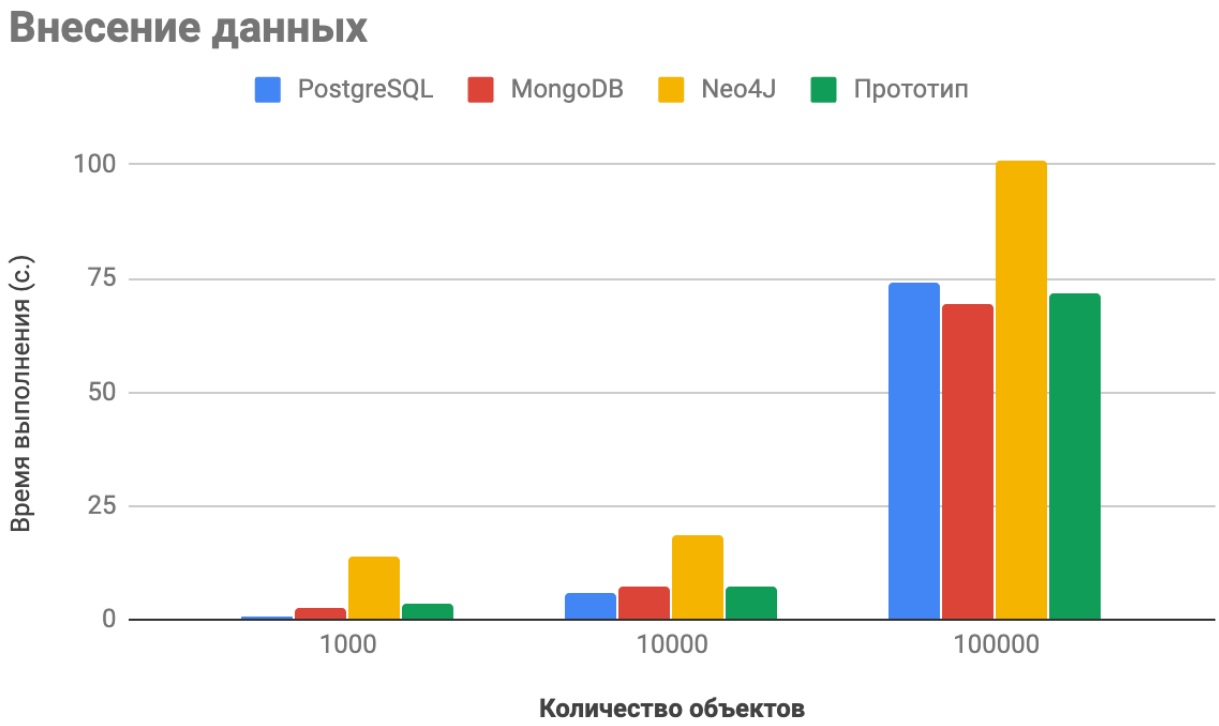

Рисунок 6 – Результаты выполнения запросов на получение данных

Как видно из графиков, использование разработанного модуля не вносит существенного влияния на производительность запросов в рамках одной базы данных. Иначе говоря, она зависит только от выбранной модели хранилища и не

отличается от варианта, когда это хранилище использовалось бы в качестве единственного.

4.2. Мультимодельные запросы

Данный эксперимент направлен на сравнение двух подходов:

1. Данные хранятся в единственном хранилище.
2. Разные типы данных хранятся в разных хранилищах, межмодельные связи обеспечиваются разработанным инструментом.

Для этого был разработан сценарий, который включает как добавление, так и получение данных. До проведения замеров были созданы 10000 объектов типа A и 5000 объектов типа B. Объекты типа A являются сильносвязными (7000 связей) и при использовании подхода №2 хранятся в графовой базе данных.

Сценарий:

1. Добавить 3000 связей между объектами типов A и B.
2. Запросить 100 объектов типа B, затем связанные с ними объекты типа A и их «соседей».

Ниже представлены результаты выполнения каждого этапа.

Добавление связей (шаг 1).

Таблица 3 – Результаты выполнения запросов на добавление связей, в секундах

PostgreSQL	MongoDB	Neo4J	Прототип
2,7	4,7	3,8	4,5

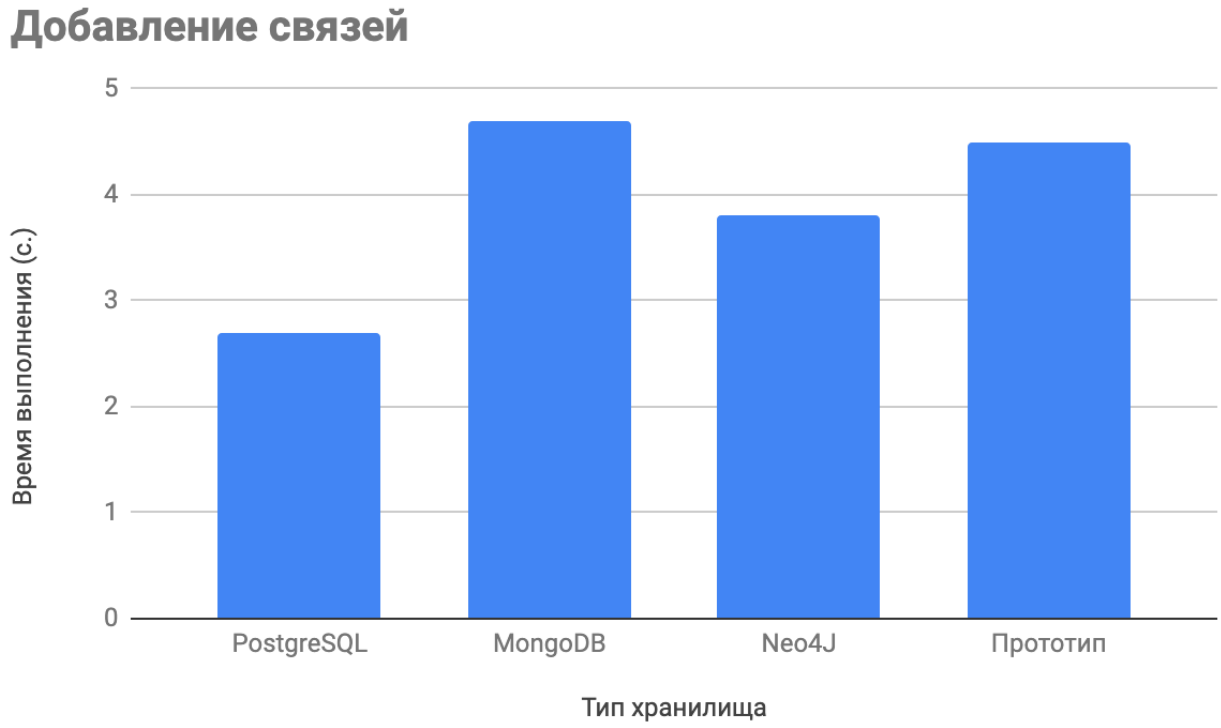

Рисунок 7 – Результаты выполнения запроса на добавление связей

Получение связанных данных (шаг 2.)

Данный запрос в большей степени относится к графовой модели, но в силу того, что заранее известна глубина рекурсивного поиска, он также может был выполнен и в других моделях. Результат содержал идентификатор полученного объекта, одно текстовое и одно числовое поле. Из 100 исходных объектов было получено 84,544 «соседей» и «соседей-соседей». При использовании MongoDB поиск осуществлялся при помощи Aggregation Framework.

Таблица 3 – Результаты выполнения запросов на добавление связей, в секундах.

PostgreSQL	MongoDB	Neo4J	Прототип
5,3	9	2,5	3,2

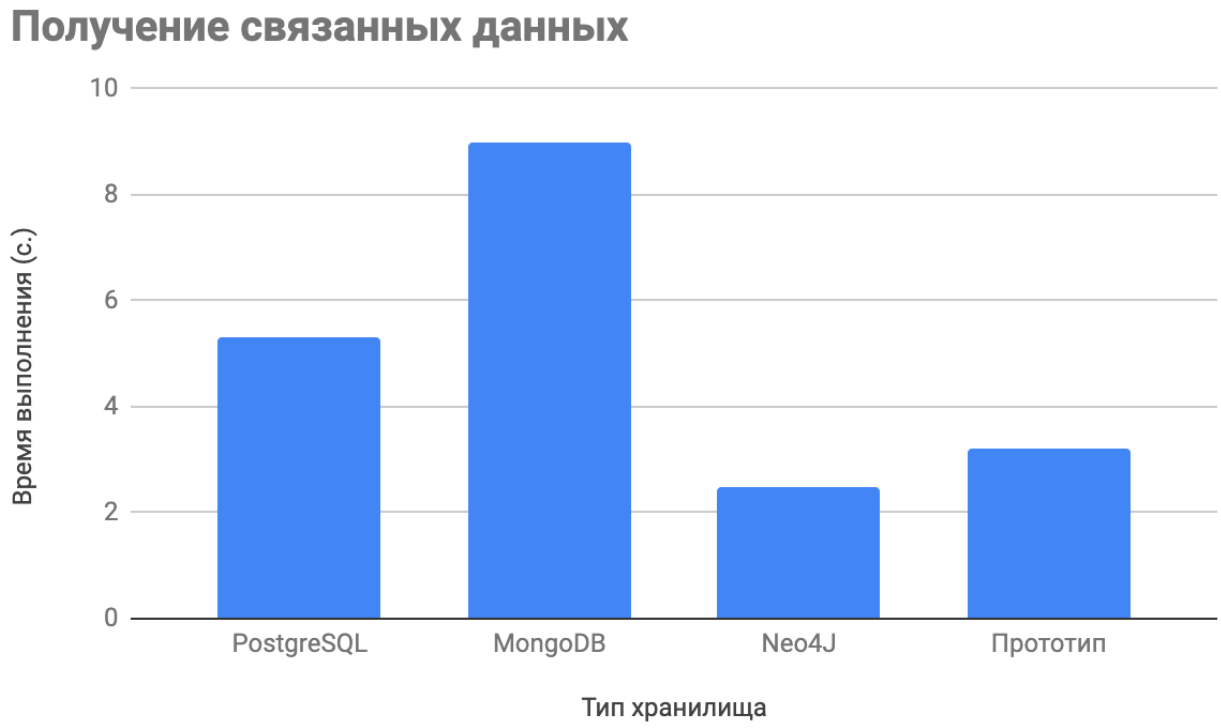

Рисунок 8 – Результаты выполнения запроса на добавление связей

На графиках можно видеть рост производительности выполнения подобного запроса при хранении данных в реляционной и документоориентированной базе данных, а задержка, которая возникает при добавлении связей, меньше, чем разница во время их получения. При этом были использованы преимущества как реляционной, так и графовой модели.

4.3. Специфичные для разных моделей запросы

Разработанный модуль также позволяет выполнять специфичные для конкретных моделей запросы. Например, поиск **кратчайшего пути** между двумя объектами. Такой тест присущ графовым моделям и имеет хорошую производительность в соответствующих базах данных. В рамках эксперимента было выполнено 1000 запросов на получение кратчайшего пути между двумя объектами (объекты менялись каждую итерацию). В силу сильной связности хранимых данных, такой запрос является достаточно сложным, так как количество соседей растет экспоненциально с ростом радиуса поиска. Поиск

кратчайшего пути имеет плохую производительность в базах данных других моделей из-за того, что содержит априори неизвестную длину этого пути, которое приводит к такому же число объединений (joins). Длина самого длинного кратчайшего пути между двумя объектами в тестовых данных равна 11.

В результате, только хранение в графовой базе данных показало приемлемый результат. Для реляционной БД такой запрос занял более чем в 100 раз больше времени, а для документоориентированной не выполнился вообще.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были проанализированы различные модели для хранения данных, а также подходы к взаимодействию с данными многих моделей.

На основе подхода Polyglot Persistence был спроектирован и разработан модуль, который позволяет использовать разные хранилища для разных видов данные и при этом хранить информацию о межмодельных связях. Основой данного модуля служит набор методов, который предоставляет веб-фреймворк Django, что, в свою очередь, позволяет разработчикам при разработке приложений взаимодействовать с базами данных разных моделей так же, как и с реляционными и не вносить изменения в уже функционирующие системы.

Также, был проведен сравнительный анализ производительности при использовании баз данных одной и многих моделей, который показал, что при необходимости выполнения специфичных запросов, которые зависят от модели хранилища, разработанный инструмент позволяет добиться меньшего времени их выполнениях. При этом данный модуль не влияет на производительность при взаимодействии с объектами внутри одного хранилища. Кроме того, модуль предоставляет возможность, разбивая данные на разные модели, выполнять осуществлять взаимодействие, которое не

В дальнейшем планируется расширить функциональность разработанного модуля и добавить следующие функции:

- возможность соединения объектов двух классов разными типами связи;
- реализация сервисного хранилища для межмодельных связей с помощью реляционной или документоориентированной базы данных;
- возможно хранения внутри сущности связи из сервисного хранилища дополнительной информации;
- автоматическое удаление из сервисного хранилища объектов, которые не имеют действующих связей.

Исходный код разработанного модуля доступен по ссылке http://gititis.kpfu.ru/Timerhanov/django-cdbms. Результаты данной работы были представлены на конференции “Электронная Казань 2019” и опубликованы в статьях [20] [21].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Lee I. Big data: Dimensions, evolution, impacts, and challenges // Business Horizons. 2017. Vol. 60. No. 3. pp. 293-303.
2. DB-Engines Ranking 2019. [Электронный ресурс] [2019]. URL: https://db-engines.com/en/ranking, (дата обращения: 28.05.2019).
3. Abramova V., Bernardino J., Furtado P. Experimental evaluation of NoSQL databases // International Journal of Database Management Systems. 2014. Vol. 6. No. 3. P. 1.
4. Sadalage P.J., Fowler M. NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education, 2012.
5. James B.E., Asagba P.O. Hybrid database system for big data storage and management // International Journal of Computer Science, Engineering and Applications (IJCSEA). 2017. Vol. 7. No. 3/4. pp. 15-27.
6. Wu C., Zhu Q., Zhang Y., Du Z., Ye X., Qin H., Zhou Y. A NOSQL–SQL hybrid organization and management approach for real-time geospatial data: A case study of public security video surveillance // International Journal of Database Management Systems. 2017. Vol. 6. No. 1. pp. 21 - 36.
7. Yan C., Cheung A., Yang J. Understanding database performance inefficiencies in real-world web applications // Proceedings of the 2017 ACM on Conference on Information and Knowledge Management. 2017. Vol. 6. No. 2. pp. 1299-1308.
8. Codd F.E. A Relational Model of Data for Large Shared Data Banks // Information Retrieval. 1970. Vol. 13. No. 6. pp. 377-387.
9. Deepak G. A Critical Comparison of NOSQL Databases in the Context of Acid and Base. St. Cloud: St. Cloud State University, 2016.
10. Indrawan-Santiago M. Database Research: Are We at a Crossroad? Reflection on NoSQL // Proceedings of the 15th International Conference on Network-Based Information Systems. 2012. pp. 45 - 51.

11. Makris A. A classification of NoSQL data stores based on key design characteristics. // Procedia Computer Science. 2017. Vol. 97. pp. 94 - 103.

12. Что такое база данных на основе пар «ключ-значение»? [Электронный ресурс] // Amazon Web Services: [сайт]. URL: https:// aws.amazon.com/ru/nosql/key-value/ (дата обращения: 27.05.2019).

13. Vicknair C., Macias M., Zhao Z., Nan X. Comparison of a graph database and a relational database: a data provenance perspective // Proceedings of the 48th Annual Southeast Regional Conference. 2010.

14. De Marzi M. Neo4J is faster than MySQL in performing recursive query // Graphs with Neo4. 2017. URL: https://maxdemarzi.com/2017/02/ 06/neo4j-is-faster-than-mysql-in-performing-recursive-query/ (дата обращения: 22.05.2019).

15. Mohamed A., Obay G., Mohammed O. Relational vs. NoSQL Databases: A Survey // International Journal of Computer and Information Technology. 2014. Vol. 3. No. 3. pp. 598 - 601.

16. Hashem I., Yaqoob I. The rise of “big data” on cloud computing: Review and open research issues // Information Systems. 2015. Vol. 47. pp. 98-115.

17. Мультимодельные базы данных [Электронный ресурс] // Национальная библиотека им. Н. Э. Баумана: [сайт]. [2017]. URL: https://ru.bmstu.wiki/Мультимодельные_базы_данных (дата обращения: 29.05.2019).

18. Models and databases [Электронный ресурс] // Django Framework Documentation: [сайт]. [2019]. URL: https://docs.djangoproject.com/en/ 2.2/topics/db/ (дата обращения: 03.05.2019).

19. Metaprogramming in Python [Электронный ресурс] // IBM Developer: [сайт]. [2018]. URL: https://developer.ibm.com/tutorials/ba-metaprogramming-python/ (дата обращения: 01.06.2019).

20. Абрамский М., Тимерханов Т. Сравнительный анализ использования реляционных и графовых баз данных в разработке цифровых образовательных систем // Вестник Новосибирского государственного университета. Серия: Информационные технологии. 2018. Т. 16. № 4. С. 5-12.
21. Тимерханов Т., Абрамский М. Механизмы совместного использования реляционных и нереляционных баз данных в веб-фреймворках // Ученые записки института социальных и гуманитарных знаний. 2019. Т. 17. № 1. С. 617 - 623.

ПРИЛОЖЕНИЕ

Исходный код приложения

Определение классов для поддержки межмодельных связей

```
from typing import Any

from django.db.models.base import ModelBase
from neomodel import NodeMeta

from crossdb.relationship import CrossDBRelationship


class CrossDBModelMeta(ModelBase,
                       NodeMeta):
    def __new__(mcls, name, bases, attrs):
        mcls.cross_db = True

        all_rels = mcls.defined_cross_relationships(attrs)
        # mcls.__all_cross_relationships__ = all_rels
        attrs['__all_cross_relationships__'] = all_rels

        return super(CrossDBModelMeta, mcls).__new__(mcls,
                                                     name,
                                                     bases,
                                                     attrs)

    @classmethod
    def defined_cross_relationships(cls, attrs, rels=False):
        props = {}
        for name, property in attrs.items():
            if isinstance(property, CrossDBRelationship):
                props[name] = property
        return dict(props)


class CrossDBModel(metaclass=CrossDBModelMeta):
    GENERATED_ID_FIELDS: dict
    _data = dict()

    def __new__(cls, *args, **kwargs) -> Any:
```

```
            cls = super(CrossDBModel, cls).__new__(cls)

            for key, val in cls.__all_cross_relationships__.items():
                cls.__dict__[key] = val.build_cross_manager(cls, key)
            return cls

        def __init__(self, *args, **kwargs) -> None:
            super().__init__(*args, **kwargs)
```

Межмодельные связи

```
    import sys

    from past.types import basestring

    from crossdb.lib import import_module
    from crossdb.manager import RelationalManager, GraphManager,
DocumentManager

    INNER = 0
    OUTER = 1


    class CrossDBRelationship:

        def __init__(self, _raw_class,
                     field_name: str,
                     cls_path: str,
                     strategy: int,
                     related: bool = False,
                     manager=RelationalManager):

            super().__init__(cls_path, field_name)
            self._raw_class = _raw_class
            self.module_name = sys._getframe(4).f_globals['__name__']
            self.manager = manager
            self.definition = {
                'field_name': field_name,
                'related': related,
                'strategy': strategy
            }
            if '__file__' in sys._getframe(4).f_globals:
                self.module_file = sys._getframe(4).f_globals['__file__']
```

```
            self.target_class = self._lookup_target_class()

        def _lookup_target_class(self):
            if not isinstance(self._raw_class, basestring):
                return self._raw_class
            else:
                name = self._raw_class
                if name.find('.') == -1:
                    module = self.module_name
                else:
                    module, _, name = name.rpartition('.')

                if module not in sys.modules:
                    # yet another hack to get around python semantics
                    # __name__ is the namespace of the parent module for
                    #__init__.py files,
                    # and the namespace of the current module for other .py
                    # files,
                    # therefore there's a need to define the namespace
                    # differently for
                    # these two cases in order for . in relative imports to
work
                    # correctly
                    # (i.e. to mean the same thing for both cases).
                    # For example in the comments below, namespace == myapp,
                    # always
                    if not hasattr(self, 'module_file'):
                        raise ImportError("Couldn't lookup
'{0}'".format(name))

                    if '__init__.py' in self.module_file:
                        # e.g. myapp/__init__.py -[__name__]-> myapp
                        namespace = self.module_name
                    else:
                        # e.g. myapp/models.py -[__name__]-> myapp.models
                        namespace = self.module_name.rpartition('.')[0]

                    # load a module from a namespace (e.g. models from myapp)
                    if module:
                        module = import_module(module, namespace).__name__
                    # load the namespace itself (e.g. myapp)
```

```
                    # (otherwise it would look like import . from myapp)
                    else:
                        module = import_module(namespace).__name__
                return getattr(sys.modules[module], name)

        def build_cross_manager(self, source, name):
            return self.manager(source,
                                name,
                                self.definition,
                                self.target_class)


    class RelationRelationship(CrossDBRelationship):

        def __init__(self,
                     _raw_class,
                     field_name: str,
                     cls_path: str,
                     strategy: int,
                     related: bool = False,
                     manager=RelationalManager):
            super().__init__(_raw_class, field_name, cls_path, strategy,
related, manager)


    class GraphRelationship(CrossDBRelationship):
        def __init__(self,
                     _raw_class,
                     field_name: str,
                     cls_path: str,
                     strategy: int,
                     related: bool = False,
                     manager=GraphManager):
            super().__init__(_raw_class, field_name, cls_path, strategy,
related, manager)


    class DocumentRelationship(CrossDBRelationship):
        def __init__(self,
                     _raw_class,
                     field_name: str,
                     cls_path: str,
```

```
                        strategy: int,
                        related: bool = False,
                        manager=DocumentManager):
                super().__init__(_raw_class, field_name, cls_path, strategy,
related, manager)
```

Классы-менеджеры

```
    from crossdb.relationship import INNER, OUTER
    from .lib import get_service_database


    class RelationshipManager(object):
        objects_field = ''

        def __init__(self, source, key, definition, target_class):
            self.source = source
            self.source_class = source.__class__
            self.name = key
            self.definition = definition
            self.target_class = target_class

            if self.definition['strategy'] == INNER:
                self._db = get_service_database()

        def all(self):
            return_set = []
            objects = getattr(self.target_class, self.objects_field)

            if self.definition['strategy'] == INNER:
                fld = getattr(self.source, self.definition['field_name'])
                return_set = objects.filter(pk__in=fld)
            elif self.definition['definition'] == OUTER:
                p_label, c_label = self.source_class.__name__,
self.target_class.__name__

                query = f"""
                MATCH (parent:{p_label})-[HAS]->(child:{c_label})
                WHERE parent.uid={self.source.uid}
                RETURN DISTINCT collect(child.uid)
                """
                result = self._db.cypher_query(query)
```

```
                return_set = objects.filter(uid__in=result[0][0])
            return return_set

        def connect(self, target_instance):
            if self.definition['strategy'] == INNER:
                setattr(self.source,
                        self.definition['field_name'],
                        target_instance.uid)
                return True
            elif self.definition['definition'] == OUTER:
                p_label, c_label = self.source_class.__name__,
self.target_class.__name__

                query = f"""
                MATCH (child:{p_label}) WHERE c.uid={target_instance.uid}
                MATCH (parent:{c_label}) WHERE p.uid={self.source.uid}
                MERGE (parent)-[:HAS]->(child)
                """
                self._db.cypher_query(query)
                return True

            return False

        def delete(self, target_instance):
            if self.definition['strategy'] == INNER:
                fld = getattr(self.source, self.definition['field_name'])
                fld.remove(target_instance.uid)
                return True
            elif self.definition['definition'] == OUTER:
                p_label, c_label = self.source_class.__name__,
self.target_class.__name__

                query = f"""
                MATCH (child:{p_label}) WHERE c.uid={target_instance.uid}
                MATCH (parent:{c_label}) WHERE p.uid={self.source.uid}
                MATCH (parent)-[rel:HAS]->(child)
                DELETE rel
                """
                self._db.cypher_query(query)
                return True

            return False
```

```
class RelationalManager(RelationshipManager):
    objects_field = 'objects'


class GraphManager(RelationshipManager):
    objects_field = 'nodes'


class DocumentManager(RelationshipManager):
    objects_field = 'objects'
```