САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет
Кафедра системного программирования

Шапошников Алексей Игоревич

Интеграция Hazelcast в сервер федеративных запросов

Производственная практика

Научный руководитель:
ст. преп. С. Ю. Сартасов

Научный консультант:
директор департамента высокотехнологичного производства ООО “БФТ”
Б. В. Щукин

Санкт-Петербург 2021 г.

Содержание

Введение

С каждым годом становится все больше различной информации — равно, как и людей, ответственных за работу с разрастающимися базами данных, их поддержку и улучшение производительности. Но у вертикального улучшения производительности существует потолок, после которого добавление новых мощностей дает незначительный эффект – тогда появляется смысл в горизонтальном расширении.

Можно делать базы данных распределенными, балансировать нагрузку на сервера, доступ к удаленной базе все равно зачастую являются узким местом системы. Поэтому появилось кэширование – сохранение последних запросов или конкретных таблиц в оперативной памяти, чтобы для популярных поисковых запросов и сущностей не обращаться к базе данных. Однако и у кэширования несложно найти недостатки: даже если иметь мощные сервера с большим количеством предзагруженных в кэш данных (что называется "прогрев"), то по этим данным нельзя строить сложные запросы с фильтрами, потому что обычный кэш не гарантирует хранение всех данных конкретной сущности и что они не успели устареть.

Эта проблема решается паттернами доступа к источникам данных: write-through (запись в промежуточный узел – например, кэш – вызывает автоматическую запись в источник данных), write-behind (то же самое, но асинхронно), read-through (если на конкретный запрос не известно, хранится сущность или нет, то промежуточное звено обращается к источнику данных и запоминает результат) и другими вендор-специфическими шаблонами. Но хранить всю информацию на одном сервере, тем более, в оперативной памяти, может стать слишком большой нагрузкой, поэтому важна возможность распределять информацию по различным узлам, которые смогут координированно и консистентно ею управлять; и раз есть запрос на обработку более сложных запросов, чем поиск всех сущностей или поиск по id, то было бы полезно, если бы технология позволяла обрабатывать сохраненные в оперативной памяти данные.

На основе этих идей было сформулировано определение IMDG [1] (In-Memory Data Grid, data grid) – это набор географически распределенных компьютеров или сервисов, которые могут взаимодействовать друг с другом и осуществлять координацию большого количества информации. Такой подход предполагает хранение данных в оперативной памяти для более

быстрого доступа и работы с ними.

В компании ООО "Бюджетные и финансовые технологии" разрабатывается ORM-фреймворк DataMaps (с планами на выпуск в open-source). В рамках отдельного модуля фреймворка Combinator (Federated Search Engine [2], FSE [3], сервер федеративных запросов – технология, позволяющая искать разные сетевые ресурсы, пользуясь одним интерфейсом) была поставлена задача использования технологии data grid для улучшения пользовательского опыта взаимодействия с продуктами.

Цели и задачи

Для оптимизации пользовательского опыта взаимодействия с продуктом, улучшения скорости запросов внедрить в библиотеку компании технологию IMDG. Для этой задачи можно выделить отдельные подзадачи:

- Провести обзор технологий IMDG и выбрать подходящий для поставленной задачи data grid;
- Внедрить технологию во внутренний продукт компании и проверить функциональность на специфических классах разрабатываемой ORM DataMaps;
- Реализовать шаблоны коммуникации с базой данных (read-through, write-through), выгрузку из базы данных;
- Реализовать модуль источника данных для распределенных запросов с использованием data grid (data grid resolver, разрешатель) для ORM, который будет определять, как получить данные по запросу в зависимости от условий в запросе и располагаемых на узле сущностей:
  - Завести источник "правил" для разрешателя для конфигурации его поведения;
  - Научиться при создании экземпляра разрешателя находить и запоминать информацию о существующих в кластере распределенных объектах;
  - Анализировать запрос на фильтры – поддерживать запросы на невложенные поля сущности, либо поля в представлении json;
  - Поддерживать запросы с конкретными ключами (или наборами ключей), лимитами и офсетами, сортировкой;
- Протестировать работу разрешателя в сервере федеративных запросов в автоматизированных тестах и на реальном продукте, а также провести замеры сравнения с запросами к удаленной базе данных и сделать вывод о возможности дальнейшего использования технологии и ее преимуществах.

Обзорная часть

Перед рассмотрением самой работы введём основные термины и определения

Обзор внутреннего ORM-фреймворка

DataMaps — это легковесный ORM-фреймворк, разработанный на языке Kotlin. Он позволяет осуществлять стандартные ORM-операции: формирование запросов к БД, представление табличных строк в виде объектов, фиксацию изменений в БД. DataMaps готовится к выпуску на рынок в качестве open-source продукта.

DataMaps разработан поверх Spring Framework [4] и его низкоуровневой реализации JDBC [5] – Java Database Connectivity [6], спецификации подключения к базам данных в Java. Spring [7] – самый популярный фреймворк для разработки промышленных приложений на Java (с поддержкой других языков семейства JVM), который ввиду своей легкости в использовании, гибкой функциональности и широкой поддержки занимает лидирующие позиции в Java-сообществе; поэтому ORM-фреймворк на платформе Spring имеет достаточно широкий целевой рынок.

Карты данных (Data Maps) – это специализированные объекты для представления табличных сущностей. Они не являются классическими POJO [8] (Plain Old Java Object), обычными Java объектами, не имеющими вешних ограничений, как наследование или реализация, и имеют собственное внутреннее устройство на основе пар «ключ-значение».

Каждая карта данных имеет свойство *id* (уникальный идентификатор) и *entity* (соответствует имени таблицы БД). Кроме того, карта данных может иметь следующие типы свойств:

- скалярные свойства – простые типы;
- ссылочные свойства – свойство содержит ссылку на другую карту данных (фактически, на другую сущность);
- списочные свойства – свойство содержит список других карт данных (фактически, коллекцию других сущностей).

Наборы полей (Field Sets) — это классы-шаблоны для построения карт данных с нужной структурой полей. Для одной табличной сущности можно создать произвольное количество наборов полей. Объекты наборов полей

не только позволяют конструировать карты данных, но и предлагают удобный строго-типизированный DSL для построения запросов к БД.

Проекции (Projections) — это объекты, инкапсулирующие запрос к БД. Если наборы полей задают структуру и состав полей, которые в принципе возможно загрузить из БД, используя конкретный набор полей, то проекции определяют, какие именно физические данные (т.е. какие конкретно колонки и строки таблицы) будут загружены в карты данных.

Таким образом, в наборе полей может быть задано любое подмножество колонок таблицы, а в проекции, в свою очередь, может быть задано любое подмножество полей соответствующего набора полей.

Сервис данных (DataService) – это основной сервис фреймворка DataMaps. Он регистрируется как bean [9] (бины – основные объекты приложения, которые позволяют Spring реализовывать инверсию контроля и внедрение зависимостей [10]) – и предлагает исчерпывающий API для работы с картами данных, проекциями и сущностями.

Combinator – это реализация Federated Search Engine, сервера федеративных запросов. Он позволяет получать доступ к разным сетевым ресурсам, используя один интерфейс.

С ростом распространенности больших данных, систем их хранения и необходимости обрабатывать эти данные из разных источников стали более востребованы технологии, позволяющие обращаться ко множеству источников в одном запросе: Apache Drill [11], Arenadata Prostore [12], Apollo Federation [13], Amazon Athena Federated Query [14], Presto [15] и многие другие. У этих технологий есть отличия в целевой аудитории, сценариях использования, функциональности, но их всех объединяет умение централизованно запрашивать различные источники данных и возвращать цельный ответ.

Учитывая, что в DataMaps нет сильной связности с схемами табличных сущностей (вместо них используются FieldSets), есть поддержка разных диалектов SQL (postgreSQL, Oracle SQL, HSQLDB, Firebird, MySQL), а DSL (domain-specific language) для построения запросов к БД универсален и не зависит от используемого диалекта, то было решено, что расширение фреймворка с помощью сервера федеративных запросов заметно улучшит пользовательский опыт и функциональность продукта.

На рис. 1 Combinator – сервер федеративных запросов;
PlanBuilder – компонент, отвечающий за составление плана запросов;

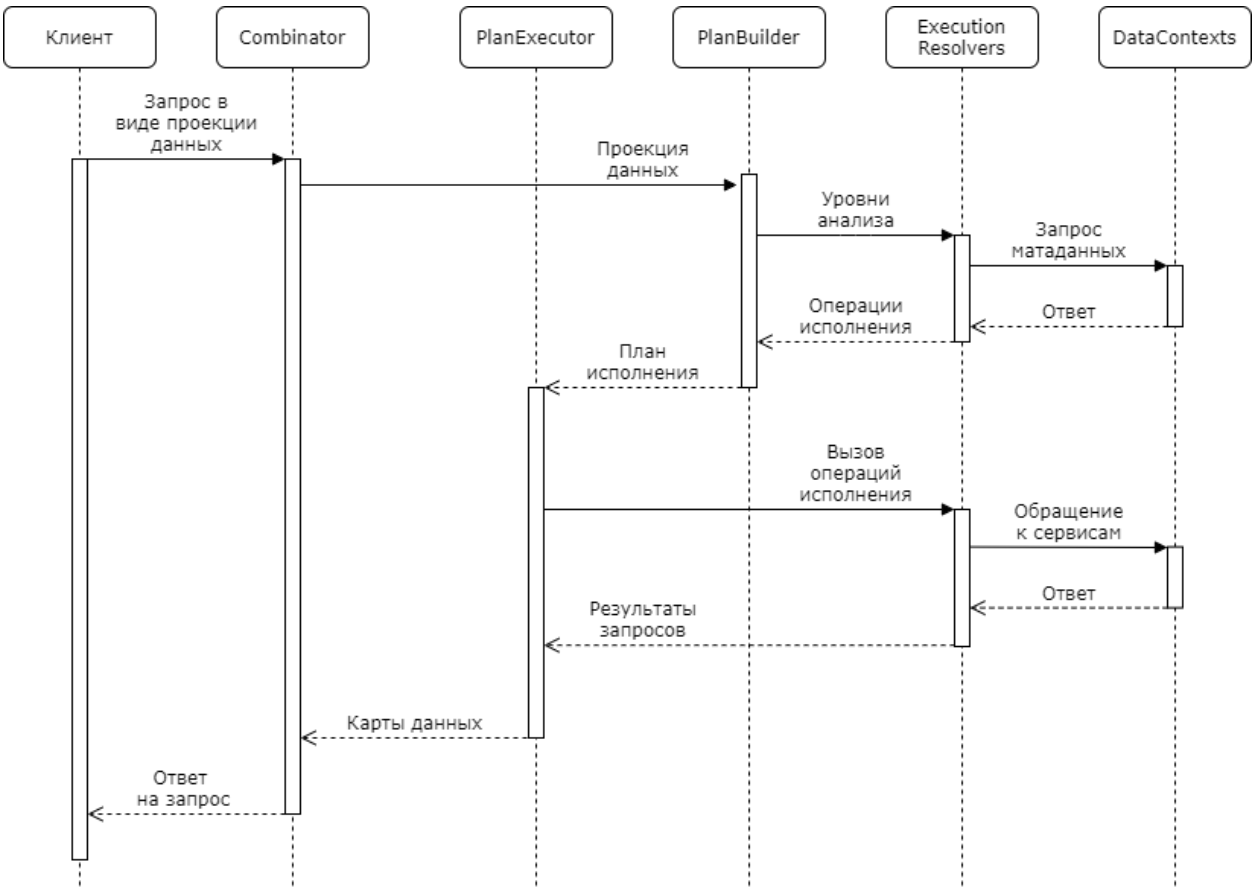

Рис. 1: Диаграмма взаимодействия сервера федеративных запросов

PlanExecutor – компонент, отвечающий за выполнения запросов из плана и объединение результатов;
ExecutionResolver – ”разрешатель”, интерфейс, отвечающий за область поддерживаемых запросов и сущностей, строящий корневые и вторичные запросы, осуществляющий присоединение ответов на вторичные запросы к родительской сущности;
DataContext – контекст источника данных, предоставляет сервисы по метаданным, построению запросов, доступа к удаленным ресурсам и т.д.

FSE работает в следующем порядке:

1. Анализ проекции запроса по уровням, какие запрашиваемые поля принадлежат каким сущностям. Например, родительская сущность Person будет высшим уровнем, а дочерняя сущность Gender, на которую она ссылается, будет более низким уровнем;

2. Сортировка источников данных по их "заинтересованности"в рассматриваемых сущностях – по приоритету того, к кому более целесообразно

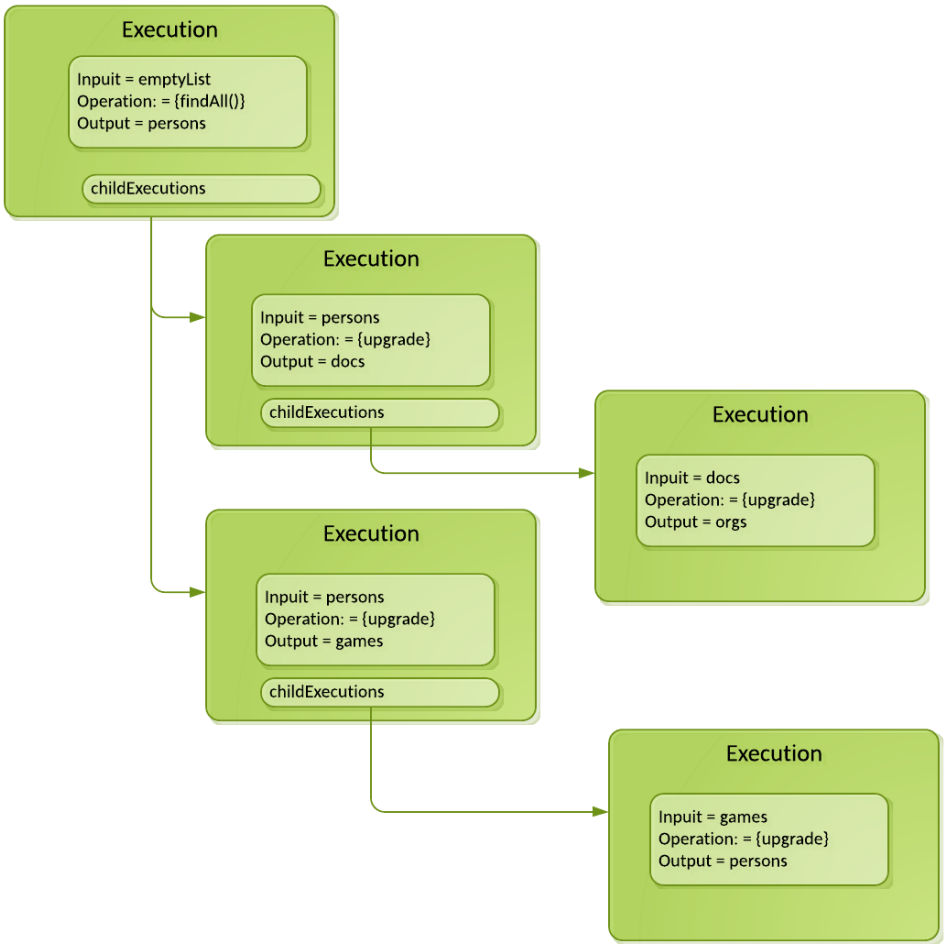

Рис. 2: Представление разбиения запроса на дерево исполнений

обратиться. Например, кэшированные данные будут иметь более высокий приоритет, чем данные в БД, но кэш имеет нулевой приоритет для запроса с фильтром, который не может обработать;

3. Построение плана исполнения запроса в виде дерева исполнений (см. рисунок 2) , у которого сущность верхнего уровня является корнем, а вложенные сущности представляются в виде детей – узлов исполнения. План отвечает за то, какому разрешателю соответствует данный уровень дерева, какие есть фильтры и опции, какие дочерние исполнения;

4. Исполнение всех запросов – начиная с корня дерева, подается запрос разрешателю с наивысшим приоритетом, затем полученная карта дан-

ных передается во все дочерние узлы исполнения и те рекурсивно исполняют свои запросы, опираясь на внешние ключи полученных данных. При этом понятно, что если у узла исполнения есть несколько детей, то это независимые сущности и запросы можно исполнять параллельно;

5. Присоединение найденных карт данных в обратном порядке исполнения – начиная с конца дерева, карты данных присоединяются по ссылочным ключам; затем данные сортируются и возвращается ответ в виде набора карт данных.

На момент написания работы в сервере федеративных запросов было реализовано три разрешателя:

- MultipleDBResolver – распознает отдельные сущности в конкретной БД, учитывают связи сущности в других базах; принимает контекст источника данных, с помощью которого запрашивает метаданные и исполняет запросы;
- EntityCacheResolver – кэш сущностей, ссылок и коллекций в сущностях;
- QueryCacheResolver – кэш с ключами в виде конкретных запросов – проекций – и значениями в виде ответов.

IMDG & IMDB

Есть схожее понятие In-Memory Database [16] (IMDB, резидентная база данных) – это СУБД, которая в первую очередь использует память, а не дискового пространства; IMDB разработаны для минимального времени ответа благодаря отсутствию необходимости обращаться к диску, что влечет за собой определенные риски сохранности данных. По этим причинам резидентные базы данных используют в системах, где важно время отклика, например, телекоммуникационные сети, торги в реальном времени и других. Основные отличия IMDG и IMDB – это:

- Резидентные СУБД функционально являются обычными RDBMS (реляционными СУБД), размещенными в пространстве оперативной памяти – что влечет за собой способ взаимодействия, полную поддержку SQL и подобное; IMDG же большей частью представляют собой распределенные карты "ключ-значение"и не всегда имеют хорошую поддержку SQL, взамен предоставляя возможности для MPP [17] (Massively

Parallel Processing), т.е. распределения данных в виде кластеров и более быстрой параллельной их обработки;

- Из этого вытекает следующие отличие – data grid позволяет естественным образом расширяться до сотен и тысяч узлов благодаря массово-параллельной архитектуре; в то же время IMDB по определению СУБД имеет ограничение на масштабируемость из-за операций соединения. При этом топология в виде единственного узла у обеих технологий сравнима по скорости ответов;
- IMDG работает с существующей базой данных, предоставляя прослойку массивно-распределенного хранилища памяти между приложением и базой данных; большинство data grid также позволяют реализовывать паттерны доступа (write-through, read-through), отчего возрастает интеграция с существующей базой данных – и все это требует внесения изменений в логику приложения для поддержки технологии, т.к. появляются новые интерфейсы для запросов и обработки данных;
- Резидентные СУБД в свою очередь не требуют изменений в логике для их поддержки, но замещают собой существующую базу данных (либо являются дополнительной для ускорения производительности).

IMDG & distributed cache

Чтобы лучше понять разницу технологий IMDG и кэширования, возьмем близкий к нашей теме по смыслу распределенный кэш (distributed cache [18]):

1. Распределенный кэш – кэш, располагающийся на множестве узлов; он так же работает с парами "ключ-значение"и стандартными операциями, как put/get, но еще предоставляет возможность к разбиению (partitioning), репликации и резервированию (backup) данных.
2. Data grid в определенном смысле является распределенным кэшем с возможностью обработки данных, формирования запросов и реализации шаблонов доступа к источнику данных.

Сравнение аналогов

Определившись с пониманием того, что такое data grid, перейдем к критериям, согласно которым происходил отбор среди производителей техно-

логии IMDG:

- Шаблоны доступа, как read-through, write-through и другие – т.к. актуальность применения технологии связана с возможностью использовать запросы с фильтрами, то выбранный data grid должен обеспечивать согласованность данных, сохраняя для конкретных сущностей актуальное состояние данных;
- Отсутствие связности с схемой данных – поскольку ORM-фреймворк предоставляет собственный DSL для наборов полей (FieldSet), то лучшим вариантом было бы внедрение технологии без опоры на схему базы данных, необходимости добавлять аннотации, чтобы помечать POJO классы и т.д.;
- Следует необязательный, но желательный критерий универсальности конфигурации data grid, которая позволит взаимодействовать с любыми новыми сущностями и картами данных;
- Поддержка Java;
- Необязательный, но желательный критерий SQL-подобного интерфейса построения запросов – ввиду особенностей внутренней работы платформы, исходя из конкретной проекции, сервисы предоставляют все необходимые метаданные для построения SQL-запросов к базе данных, что может быть переиспользовано для запросов к data grid;
- Возможность строить предикаты по не прописанным заранее полям документов; например, фильтр на поле типа json/jsonb (text/varchar в зависимости от диалекта), включая вложенные в json поля

Обратим внимание, что из сравнения вендоров были исключены такие популярные (согласно db-engines.com [19]), но не-подходящие по техническому заданию технологии, как:

- Redis – несмотря на возможность использования в качестве IMDB, кэша, брокера сообщений, Redis не предоставляет функциональность IMDG "из коробки";
- Amazon DynamoDB [20] – масштабируемая база данных для пар "ключ-значение"и документов, работающая без строгой типизации схемы;

Таблица 1: Таблица сравнения вендоров

Название	Memcached [22]	etcd [23]	Hazelcast [24]	Ignite [25]
Шаблоны доступа	Cache aside	–	Map Store [26] (read-though, write-through и предзагрузка данных)	Read-through, write-through, write-behind [27]
Schema-free	+	+	+	–
Распределенная обработка данных	–	–	+	+
Интерфейс предикатов	–	–	Predicate API и поддержка SQL-like запросов	Поддержка SQL запросов
Предикаты по неопределенным полям	–	–	Возможность строить запросы по json объектам и коллекциям в них, не описывая заранее структуру	Необходимо заранее прописывать структуру объектов для построения предикатов по ним

- Microsoft Azure Cosmos DB [21] – еще одна масштабируемая мультимодельная база данных, обещающая низкую задержку и хорошо-определенные модели целостности данных.

Исходя из описанных критериев, было произведено краткое сравнение самых популярных технологий в изучаемой области для выбора подходящего вендора. Поскольку все рассматриваемые технологии имеют поддержку Java, обеспечение целостности данных в каком-либо виде и распределенность данных, сравнение будет по другим пунктам: см. таблицу сравнения 1.

Из этой таблицы понятно, что по поставленным под конкретную задачу критериям больше всего подошел Hazelcast IMDG: он позволяет реализовывать различные паттерны доступа к данным; работать с распределенными объектами без задания им конкретной структуры (что позволяет создавать новые сетки данных без лишней конфигурации); предоставляет мгновенную или конечную (eventual) целостность данных на выбор пользователя; позволяет строить предикаты благодаря различным интерфейсам Predicate API и заданием фильтров с помощью синтаксиса SQL; и – что стало определяющим фактором – предоставляет обертку HazelcastJsonValue для сериализации json-объектов в сетках данных, благодаря которой возможно строить запросы с фильтрами на поля и коллекции json, не определяя заранее структуру файлов.

Практическая часть

Конфигурация Hazelcast

Любое внедрение технологии начинается с ее конфигурации и подключения (в нашем случае — HazelcastInstance в качестве Spring-бина). Для Hazelcast создается единственная конфигурация для распределенных Map словарей с названием "default", что позволяет всем впоследствии создаваемым словарям наследовать эту конфигурацию.

Эта конфигурация включает в себя реализацию MapStore с интерфейсом MapStore<Any, HazelcastJsonValue> (ключ любого non-null типа, а значение типа HazelcastJsonValue, обертки над json).

MapStore

MapStore [26] – интерфейс Hazelcast, который позволяет загружать и сохранять данные в/из распределенного словаря data grid; иными словами, этот интерфейс реализует шаблоны коммуникации с источником данных write-through, read-through (можно так же настроить асинхронную запись) и предварительную загрузку данных при создании словаря.

В реализованном для DataMaps MapStore в качестве ключей могут использоваться как целочисленный тип Long, так и обычная строка, что расширяет область использования; главное, чтобы при вызове загрузки или выгрузки объектов данных типы совпадали и не случалось дупликации данных или не нахождения. Как было сказано в обзоре фреймворка DataMaps, данные представляются в виде карт данных; и при их выгрузке в распределенный словарь Hazelcast они преобразуются в формат json (с сохранением системных данных для восстановления сущности), а затем оборачиваются в HazelcastJsonValue; при выгрузке из словаря данные преобразуются в строку (json), а потом по ней собирается карта данных.

Точки взаимодействия с словарями

В этом пункте мы рассмотрим API, который используется при работе с распределенными объектами Hazelcast в разрешателе для Combinator.

Словари создаются при помощи метода класса *HazelcastInstance* `getMap<Key, Value>(name: String)`, в нашей реализации в качестве аргумента *name* передаются исключительно названия требуемых реляционных таблиц, т.к.

по названию происходит асинхронная загрузка данных данной таблицы из базы данных PostgreSQL.

Разрешатель для сетки данных Hazelcast создается при запуске приложения, поэтому на случай отказа сервера с приложением (но не отказа распределенной сети data grid) при инициализации экземпляру клиента Hazelcast подается запрос на все распределенные объекты, среди которых находятся реализации словарей, а они уже запоминается в отдельном словаре (сопоставляющем названию распределенного словаря его объект). Такое хранение всех объектов распределенных карт позволяет при анализе входящего запроса узнать, какие сущности доступны из data grid.

Единственное, что осталось отметить из используемого интерфейса Hazelcast – это предикаты для построения запросов.

Запросы в data grid

Разрешатель содержит различную логику запросов к распределенным словарям, которая определяется входящим запросом — проекцией данных.

При запросе сущности по идентификатору используется стандартный интерфейс взаимодействия с словарями `map.get(id)` или более лаконично на языке Kotlin `map[id]`. Полученный результат в виде HazelcastJsonValue преобразуется в карту данных, которая "обрезается", оставляя только запрошенные клиентом поля (это действие проделывается во всех запросах и списках результатов, поэтому дальше опускается).

При запросе сущности с фильтрами они трансформируется в SQL-подобную строку запроса (подразумевается часть sql запросов, которая идет после ключевого слова **where**). В ORM DataMaps фильтры представлены в виде абстрактного синтаксического дерева выражений, поэтому в строковое представление оно преобразуется с помощью свертки. Затем с помощью методов `Predicates.sql<Key, Value>(sqlPredicate: String)` и `map.values(predicate: Predicate)` запрашиваются подходящие под условия данные.

Для обработки запросов к данным по страницам используется интерфейс **PagingPredicate** [28], который позволяет помимо предиката-фильтра указывать количество необходимых результатов и отступ для удовлетворяющих данных. Важно, что данные в распределенных объектах лежат в случайном порядке (в отличие от стандартных баз данных, имеющих различные индексы для естественной сортировки результатов), и если ни ключ,

ни значение словаря не наследуют интерфейс `Comparable` (что справедливо при работе с DataMaps), то при одном и том же запросе могут вернуться разные данные; также очевидно, что клиенты должны уметь задавать поля, по которым сортируются результаты, поэтому возникла необходимость реализации компаратора с типом `Comparator<Map.Entry<Any, HazelcastJsonValue»`.

Реализованный компаратор, прежде всего, преобразует ключ словаря к типу **Long** или, при невозможности, к **String** – и при отсутствии указания поля для сортировки, она осуществляются по этим ключам. В противном случае HazelcastJsonValue десериализуются в карты данных, по заданным полям которых и происходит сравнение.

- Согласно контрактам Hazelcast, компаратор должен поставляться через фабрику, причем и фабрика, и возвращаемый ею компаратор должны реализовать интерфейс **Serializable**;
- Процесс сортировки данных в контексте использования data grid является очень дорогостоящей по времени операцией — после параллельной обработки данных согласно фильтрам все подходящие результаты десериализуются, затем в нашем случае конвертируются в карты данных и сравниваются. Это занимает на порядки больше времени, чем запросы без пагинации – в частности, запрос на возвращение всех результатов с фильтром отработает быстрее, чем запрос на возвращение 20 результатов с тем же самым фильтром.

Замеры производительности

Чтобы оценить полезность использования реализованного data grid модуля Hazelcast для Combinator, было решено протестировать его на реальном проекте, использующем библиотеку DataMaps, и сравнить производительность запросов по сравнению со стандартным обращением к базе данных на PostgreSQL. В качестве тестовых данных были выгружены справочники с портала открытых данных правительства Москвы [29] справочник по театрам [30] и регистр объектов культурного наследия [31].

На момент проведения замеров данных по справочнику Театры хранилось 83 записи (7 кб данных), а по Объектам культурного наследия – 8532 записи (12 мб данных); если учесть, что записей во втором справочнике примерно в 102 раза больше, то средняя запись по театру в объеме меньше

Таблица 2: Замеры времени запросов

Справочник	Театры (500 повторов на запрос)			Объекты культурного наследия (100 повторов)		
Способ запроса	Hazelcast: бинарный формат	Hazelcast: объектный формат	База данных	Hazelcast: бинарный формат	Hazelcast: объектный формат	База данных
findAll	20.084	21.976	196.148	132.97	175.25	1112.34
find с лимитом 20	1321.382	1270.958	44.256	912.73	910.96	20.76
find с лимитом 20 и фильтром	4.33	4.958	28.346	167.56	167.21	19.6
find с фильтром на табличное поле	11.654	23.89	116.96	32.82	31.13	81.32
find с фильтром на json-поле	2.616	3.256	28.924	35.85	35.65	105.7

примерно в 17 раз, чем запись об объекте культурного наследия.

Замеры проводились в качестве тестов: производились запросы к сетке данных и обычному серверу базы данных, повторяясь по 500/100 раз (для Театров и Объектов культурного наследия соответственно) на запрос для устранения погрешностей и различных не-системных факторов, время замерялось и делилось на количество прогонов. Для более точной и консистентной оценки производительности имело смысл использовать JMH [32], Java Microbenchmark Harness, однако задача состояла в приближенной к реальности оценке без лишнего внедрения технологий и траты времени.

Замеры делились на следующие категории:

- Hazelcast с бинарным форматом данных — данные хранятся в сериализованном виде;
- Объектный формат данных в памяти [33] data grid — данные хранятся в десериализованной форме, согласно документации, рекомендуется при активной работе с данными, когда большая часть взаимодействия с Hazelcast — запросы, а не get и put;
- Запрос к базе данных;

Все замеры таблицы 2 приведены в миллисекундах.

Уточним, что "find с фильтром на табличное поле" для справочника Театры возвращал 57 результатов ($\sim 70\%$ количества данных), а для Объектов культурного наследия — 499 ($\sim 5\%$); "find с фильтром на json-поле" и "find с лимитом 20 и фильтром" для Театров — 10 результатов ($\sim 12\%$) , для второго справочника — 1967 ($\sim 23\%$).

Для оценки нас интересует, когда data grid быстрее, чем стандартный

СУБД запрос, а когда медленнее; в каких случаях оправдано использование объектного типа хранения данных в Hazelcast; в заключение нас будут интересовать выводы о преимуществах data grid и актуальности в проекте со справочниками.

Как видно из таблицы, самый большой прирост data grid в скорости обработки наблюдается при запросах с фильтрами без лимитов и выгрузке всех записей; Hazelcast выигрывает по скорости обработки фильтр-запросов и на маленьком, и на большом объеме данных, а также в целом быстрее передает все эти данные, что должно быть закономерным, учитывая способ их хранения.

При этом те же запросы с фильтрами и пагинацией ухудшили показатели data grid: в случае справочника для театров показатели лучше, чем у базы данных, потому что вернувшихся результатов меньше, чем размер страницы запроса, однако в случае объектов культурного наследия Hazelcast сортировал 1967 найденных записи, чтобы вернуть 20 — и проиграл по времени тому, как этот процесс осуществляется в реляционной базе данных. Что еще более заметно по запросу "find с лимитом 20", СУБД в несколько порядков быстрее возвращает ограниченное количество результатов из выборки большого размера (что связано с внутренним устройством хранения таблиц и индексов); в то же время data grid для этого запроса десериализует и сравнивает пары результатов, чтобы отсортировать — это занимает намного больше времени, чем просто вернуть все данные.

Если сравнивать показатели data grid при бинарном и объеткном формате хранения данных, при запросе всех данных бинарный формат немного выигрывает по скорости, но и при остальных запросах с обработкой данных бинарный формат сравним с объектным. Возможно, это объясняется тем, что объектный формат хранения данных в нашем случае – строки, которые по способу взаимодействия ближе к бинарному формату, чем POJO.

Делая выводы о выгоде использования реализованного в ходе данной работы модуля Hazelcast в Combinator, можно отметить, что он справляется лучше СУБД с фильтрами (чем более сильная фильтрация, тем заметнее выигрыш по скорости) и простым запросом всех данных, однако сильно проигрывает при запросах с лимитами. Значит, data grid стоит использовать для обработки запросов, не требующих сортировки или ограничения по кол-ву возвращаемых данных.

В проекте со справочниками же все запросы, поступающие от клиен-

та для обработки серверу, содержат ограничение в 20 записей, чтобы не перегружать клиента лишней информацией, а возвращать ее равномерно с помощью пагинации. Поэтому был сделан вывод, что в данном проекте модуль data grid не представляет интереса и он был отложен в отдельную ветку для дальнейшего тестирования в других проектах.

Выводы

В рамках данной курсовой работы была выполнена цель по внедрению с Combinator технологии IMDG. В частности были выполнены следующие задачи:

- Произведен обзор технологии IMDG и выбрана подходящая под задачу реализация;
- Hazelcast подключен в качестве data grid в библиотеку DataMaps; реализованы шаблоны коммуникации write-through, read-through, выгрузка из базы, сериализация данных;
- Реализован модуль разрешателя data grid для Combinator; реализована поддержка запросов с предикатами, пагинацие, сортировками, конкретным id или множеством ключей;
- Разрешатель внедрен и протестирован на реальном проекте, замерены данные о его производительности по сравнению с реляционной базой данных, сделаны выводы о перспективах использования data grid в разных проектах.

Как стало понятно после произведения замеров времени выполнения запросов, data grid не релевантно использовать для проекта, на котором он тестировался, поскольку в нем все запросы с пагинацией. Однако от проделанной работы и полученной разработки не отказались, сохранив это в отдельной ветке на случай будущего тестирования в уже новых проектах или для справочной информации, чтобы переиспользовать логику интеграции IMDG с DataMaps.

Также к перспективам продолжения работы можно отнести изучение внутреннего устройства и оптимизаций Hazelcast для более точной оценки при замерах, проведение тестов с разными настройками сервера и топологии, сделать выводы и влиянии распараллеливания обработки запросов на затраченное время по сравнению с одним узлом-сервером, сделать выводы о том, для какие настройки сервера Hazelcast оптимальны для конкретных задач.

Список литературы

[1] Data grid. [Электронный ресурс] - https://en.wikipedia.org/wiki/Data_grid. Дата обращения 03.10.2021.

[2] Federated search. [Электронный ресурс] - https://en.wikipedia.org/wiki/Federated_search. Дата обращения 03.10.2021.

[3] Federated search engine. [Электронный ресурс] - https://www.ala.org/alcts/resources/org/cat/research/fed_search. Дата обращения 03.10.2021.

[4] Spring framework. [Электронный ресурс] - https://spring.io/projects/spring-framework. Дата обращения 03.10.2021.

[5] Spring jdbc. [Электронный ресурс] - https://spring.io/projects/spring-data-jdbc. Дата обращения 03.10.2021.

[6] Java database connectivity. [Электронный ресурс] - https://en.wikipedia.org/wiki/Java_Database_Connectivity. Дата обращения 03.10.2021.

[7] Why spring. [Электронный ресурс] - https://spring.io/why-spring. Дата обращения 03.10.2021.

[8] Plain old java object. [Электронный ресурс] - https://en.wikipedia.org/wiki/Plain_old_Java_object. Дата обращения 03.10.2021.

[9] Spring beans. [Электронный ресурс] - https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html. Дата обращения 03.10.2021.

[10] Dependency injection. [Электронный ресурс] - https://www.baeldung.com/spring-dependency-injection. Дата обращения 03.10.2021.

[11] Apache drill. [Электронный ресурс] - https://drill.apache.org/. Дата обращения 03.10.2021.

[12] Arenadata prostore. [Электронный ресурс] - https://github.com/arenadata/prostore. Дата обращения 03.10.2021.

[13] Apollo federation. [Электронный ресурс] - https://www.apollographql.com/docs/federation/. Дата обращения 03.10.2021.

[14] Amazon athena federated query. [Электронный ресурс] - https://docs.aws.amazon.com/athena/latest/ug/connect-to-a-data-source.html. Дата обращения 03.10.2021.

[15] Presto. [Электронный ресурс] - https://prestodb.io/. Дата обращения 03.10.2021.

[16] In-memory database. [Электронный ресурс] - https://en.wikipedia.org/wiki/In-memory_database. Дата обращения 03.10.2021.

[17] Massively parallel. [Электронный ресурс] - https://en.wikipedia.org/wiki/Massively_parallel. Дата обращения 03.10.2021.

[18] Distributed cache. [Электронный ресурс] - https://en.wikipedia.org/wiki/Distributed_cache. Дата обращения 03.10.2021.

[19] Db engine: key-value store rankings. [Электронный ресурс] - https://db-engines.com/en/ranking/key-value+store. Дата обращения 03.10.2021.

[20] Amazon dynamodb. [Электронный ресурс] - https://aws.amazon.com/ru/dynamodb/. Дата обращения 03.10.2021.

[21] Microsoft azure cosmos db. [Электронный ресурс] - https://azure.microsoft.com/en-us/services/cosmos-db/. Дата обращения 03.10.2021.

[22] Memcached properties. [Электронный ресурс] - https://db-engines.com/en/system/Memcached. Дата обращения 03.10.2021.

[23] Etcd properties. [Электронный ресурс] - https://db-engines.com/en/system/etcd. Дата обращения 03.10.2021.

[24] Hazelcast properties. [Электронный ресурс] - https://db-engines.com/en/system/Hazelcast. Дата обращения 03.10.2021.

[25] Ignite properties. [Электронный ресурс] - https://db-engines.com/en/system/Ignite. Дата обращения 03.10.2021.

[26] Hazelcast: Mapstore. [Электронный ресурс] - https://docs.hazelcast.com/imdg/4.2/data-structures/map.html#loading-and-storing-persistent-data. Дата обращения 03.10.2021.

[27] Ignite: шаблоны доступа. [Электронный ресурс] - https://ignite.apache.org/docs/latest/persistence/external-storage#read-through-and-write-through. Дата обращения 03.10.2021.

[28] Hazelcast predicates. [Электронный ресурс] - https://docs.hazelcast.com/hazelcast/5.0-SNAPSHOT/query/querying-maps-predicates.html#filtering-with-paging-predicates. Дата обращения 03.10.2021.

[29] Портал открытых данных. [Электронный ресурс] - https://data.mos.ru/. Дата обращения 03.10.2021.

[30] Справочник по театрам. [Электронный ресурс] - https://data.mos.ru/opendata/7702155262-teatry/passport?versionNumber=3&releaseNumber=869. Дата обращения 03.10.2021.

[31] Регистр объектов культурного наследия. [Электронный ресурс] - https://data.mos.ru/opendata/7705021556-obekty-kulturnogo-naslediya-i-vyyavlennye-obekty-kulturnogo-naslediya. Дата обращения 03.10.2021.

[32] Java microbenchmark harness. [Электронный ресурс] - http://openjdk.java.net/projects/code-tools/jmh/. Дата обращения 03.10.2021.

[33] Объектный формат данных hazelcast. [Электронный ресурс] - https://docs.hazelcast.com/hazelcast/5.0-snapshot/data-structures/setting-data-format.html. Дата обращения 03.10.2021.