**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**Белорусский государственный Университет**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра многопроцессорных систем и сетей**

ВАТОЛИН

Ростислав Павлович

**КОМПЬЮТЕРНЫЙ МОДУЛЬ КОНТРОЛЯ СИСТЕМЫ ВЕБ-СЛУЖБ НА БАЗЕ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ**

Дипломная работа

|  |  |
| --- | --- |
|  | Научный руководитель  ассистент кафедры МСС  А.С. Гусейнова |

Допущена к защите

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2017 г.

Ассистент кафедры МСС ФПМИ, А.С. Гусейнова

Минск, 2017

**АННОТАЦИЯ**

Ватолин Р.П. Компьютерный модуль контроля системы веб-служб на базе распределенных вычислений: Дипломная работа / Минск: БГУ, 2017. – 35 с.

Рассматривается задача реализации высоконагруженного модуля для контроля поступающих прогнозов от погодных веб-служб с использованием фреймворка Apache Storm.

**АНАТАЦЫЯ**

Ватолiн Р.П. Кампэютэрны модуль кантролю сiстэмы вэб-службаў на базе размеркаваных вылiчэнняў: Дыпломная праца / Мінск: БДУ, 2017. – 35с.

Разглядаецца задача рэалізацыі высаканагруженнава модуля для кантролю метэаралагічных прагнозаў ад вэб-службаў з выкарыстаннем фреймворка Apache Storm.

**ANNOTATION**

Vatolin R.P. Application for data flow control of weather web-services based on distributed calculations: Graduate work/ Minsk: BSU, 2017. – 35 p.

The problem is being considered based on implementation of highly loaded module for monitoring incoming forecasts from weather services using the Apache Storm framework.

**РЕФЕРАТ**

Дипломная работа, 35 с., 7 рис., 7 источников.

***Ключевые слова:*** МЕТЕОРОЛОГИЯ, ВЫСОКОНАГРУЖЕННОЕ ПРИЛОЖЕНИЕ, APACHE STORM, ЗАДАЧА ПРОГНОЗИРОВАНИЯ, МЕТЕОСЛУЖБА, РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ, BIG DATA.

***Объект исследования*** – метеорологические веб-сервисы, концепции и методы разработки высоконагруженных приложений, применение фреймворка Apache Storm в создании вычислительного модуля.

***Методы исследования*** – обзор и изучение фреймворка Apache Storm, исследование метеорологических сервисов, предоставляющих данные о погоде, получение теоретический и практических знаний в разработке приложений, основанных на распределенных вычислениях.

***Цель исследования***– создание компьютерного модуля с применением фреймворка Apache Storm для контроля веб-метеослужб для выявления лидера в точности прогноза.

***В результате исследования*** реализован модуль контроля веб-служб, предоставляющий достоверного источника метеорологических данных с высокой точностью, изучен и освоен ряд методов обработки огромного количества данных в реальном времени с помощью фреймворка Apache Storm.

***Область применения*** – метеороло́гия.

**РЭФЕРАТ**

Дыпломная праца, 35 с., 6 мал.,3 табліцы, 9 крыніц.

***Ключавыя словы***: МЕТЭАРАЛОГIЯ, ВЫСАКАНАГРУЖЕННАЕ ПРЫКЛАДАННЕ, APACHE STORM, ЗАДАЧА ПРАГНАЗАВАННЯ, МЕТЭОСЛУЖБА, РАЗМЕРКАВАНЫЯ ВЫЛИЧЭННI, BIG DATA.

***Аб'ект даследавання*** – метэаралагiчныя вэб-сэрвiсы, канцэпцыi i метады распрацоўкi высаканагружаных прыкладанняў, прымяненне фреймворка Apache Storm у стварэннi вылiчальнага модуля.

***Мэта даследавання*** – стварэнне камп’ютэрнага модуля з ужываннем фрэймворка Apache Storm для кантролю вэб-метэослужб для выяўлення лiдэра ў дакладнасцi прагнозу.

***У выніку даследавання*** рэалiзаваны модуль кантролю вэб-службаў, якi прадстаўляе пэуную крынiцу метэаралагiчных дадзеных з высокай дакладнасцю, вывучаны i засвоены шэраг метадаў апрацоўкi велiзарнай колькасцi дадзеных у рэальным часе з дапамогай фреймворка Apache Storm.

***Метады даследавання*** – абзор i вывуэнне фрэймворка Apache Storm, даследаванне метеаралагiчных вэб-служб, якiя прадастаўляюць дадзеныя аб надвор’i, атрыманне тэарэтычных i практычных ведаў у распрацоўцы прыкладанняў, заснаваных на размеркаваных вылiчэннях.

***Вобласць прымянення*** – метэаралогiя.

**ABSTRACT**

Thesis, 35 p., 6 fig.,3 tables, 9 sources.

***Keywords:*** METEOROLOGY, HIGHLY LOADED APPLICATION, APACHE STORM, PREDICTION PROBLEM, METEOROLOGICAL SERVICE, DISTRIBUTED CALCULATIONS, BIG DATA.

***The object of study*** – meteorological web services, concepts and methods for developing highly loaded applications, the use of the Apache Storm framework in the creation of a computational module.

***The purpose of work*** – creation of a computer module using the Apache Storm framework to monitor web meteorological services to identify the leader in the accuracy of the forecast.

***Research methods*** – the study of the Apache Storm framework, the study of meteorological services providing weather data, obtainment of theoretical and practical knowledge in the development of applications based on distributed computing.

***As a result,*** a module of web service control was created, that provide a reliable source of meteorological data with high accuracy, studied and mastered a number of methods for processing a huge amount of data in real time using the Apache Storm framework.

***Scope*** – meteorology.

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 7](#_Toc481166528)

[STORM FRAMEWORK 8](#_Toc481166529)

[**1.1** **Ключевые особенности фреймворка** 8](#_Toc481166530)

[**1.2** **Элементы Storm** 8](#_Toc481166531)

[**1.3** **Сохранность данных** 9](#_Toc481166532)

[**1.4** **Транзакции в Storm** 10](#_Toc481166533)

[EHCACHE 11](#_Toc481166534)

[**2.1 Введение в Ehcache** 11](#_Toc481166535)

[**2.2 Терминология** 12](#_Toc481166536)

[**2.3 Уровни хранения данных** 14](#_Toc481166537)

[**2.4 Улучшения производительности** 16](#_Toc481166538)

[РЕАЛИЗАЦИЯ ПОСТАВЛЕННОЙ ЗАДАЧИ 18](#_Toc481166539)

[**3.1 Получение и сохранение прогноза** 19](#_Toc481166540)

[**3.2 Получение текущего и извлечение сохраненного состояния** 21](#_Toc481166541)

[**3.3 Обработка и отчет** 23](#_Toc481166542)

[ЗАКЛЮЧЕНИЕ 27](#_Toc481166543)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 28](#_Toc481166544)

## ВВЕДЕНИЕ

Если следить за прогнозом погоды, то в итоге можно прийти к выводу, что каждый источник будет иметь свою погрешность. В температуре воздуха можно увидеть разницу в пару градусов, во влажности - до 10 процентов, а в атмосферном давлении - до 20 мм рт. ст. Если говорить об атмосферном давлении и влажности, то можно заметить, что расхождения от разных источников значительно отличаются. Неправильные данные об атмосферном давлении, могут причинить дискомфорт людям со слабыми сердечно-сосудистой, дыхательной, центральной нервной системами. Правильные данные о влажности воздуха нужны мамам с младенцами, ведь именно новорожденные очень серьезно воспринимают малейшее изменение окружающей среды.

Постоянный анализ достоверности источников позволит постоянно контролировать качество и точность предоставляемых данных. Имея постоянно работающий компьютерный модуль, который анализирует прогноз и действительные данные в разных точках как городов, так и областей, позволит сказать, какой на данный момент сервис выходит на передний план, среди остальных. Ведь одноразовый анализ и выявление лидера по качеству не будет лучшим решением поставленной задачи.

На сегодняшний день практически невозможно найти подобные механизмы анализа погодных данных. «ОПогоде» - украинский агрегатор погодных данных. Этот сервис решает похожую задачу, которая была описана выше, но имеет ряд недостатков, которые я собираюсь исправить в своем решении. Одним из основных является предоставление данных о погоде в результате усреднения нескольких сервисов, тем самым создавая данные, основываясь как на надежных источниках, так и нет.

В силу высокой загруженности данного модуля отличным решением будет использование фреймворка Apache Storm, написанного на Java и Clojure. Именно с его помощью можно будет осуществить обработку этого колоссально огромного количества данных.

**STORM FRAMEWORK**

В 2011 году Twitter открыл, под лицензией Eclipse Public License, проект

распределенных вычислений Storm. Storm был создан в компании BackType и перешел к Twitter после покупки.

Storm это система, ориентированная на распределенную обработку больших потоков данных, аналогичная Apache Hadoop, но в реальном времени.

### **Ключевые особенности фреймворка**

К ключевым особенностям фреймворка Storm относят:

1. Масштабируемость. Задачи обработки распределяются по узлам кластера и потокам на каждом узле;
2. Гарантированная защита от потери данных;
3. Простота развертывания и спровождения;
4. Восстановление после сбоев. Если какой-либо из обработчиков отказывает, задачи переадресуются на другие обработчики;
5. Возможность написания компонентов не только на Java. Простой Multilang protocol с использованием JSON объектов. Есть готовые адаптеры для языков Python, Ruby и Fancy.

### **Элементы Storm**

Основными элементами фреймворка являются:

1. Tuple. Элемент представления данных. По умолчанию может содержать Long, Integer, Short, Byte, String, Double, Float, Boolean и byte[] поля. Пользовательские типы, используемые в Tuple должны быть сериализуемыми;
2. Stream. Последовательность из Tuple. Содержит схему именования полей в Tuple;
3. Spout. Поставщик данных для Stream. Получает данные из внешних источников, формирует из них Tuple и отправляет в Stream. Может отправлять Tuple в несколько разных Stream. Есть готовые для популярных систем обмена сообщениями: RabbitMQ / AMQP, Kestrel, JMS, Kafka;
4. Bolt. Обработчик данных. На вход поступают Tuple. На выход отправляет 0 или более Tuple;
5. Topology. Совокупность элементов с описанием их взаимосвязи. Аналог MapReduce job в Hadoop. В отличии от MapReduce job — не останавливается после исчерпания входного потока данных. Осуществляет транспорт Tuple между элементами Spout и Bolt. Может запускаться локально или загружаться в Storm кластер.
6. **Сохранность данных**

Разные классы задач предъявляют различные требования к надежности. Одно дело пропустить пару записей при подсчете статистики посещений, где счет идет на сотни тысяч и особая точность не нужна. И совсем другое — потерять, например, информацию о платеже клиента.

Рассмотрим о механизмы защиты от потери данных, которые реализованы в Storm.

Если нам не важно были ли ошибки при обработке Tuple, то Spout отправляет Tuple в SpoutOutputCollector посредством вызова метода emit( new Values(...) ).

Eсли необходимо узнать успешно ли обработался Tuple, то вызов будет выглядеть как emit( new Values(...),msgId ).

где - это объект произвольного класса. В этом случае интерфейс ISpout предоставляет методы:

1. - будет вызван, если Tuple обработан успешно;
2. - будет вызван, если Tuple не обработан из-за какой-либо ошибки.

Стоит обратить внимание, что msgId — это id сообщения, с которым был вызван SpoutOutputCollector.emit. Методы nextTuple, ack и fail, вызываются в одном потоке и не требуют дополнительной синхронизации при обращении к полям Spout.

Для того что бы Bolt мог информировать Storm о результатах обработки, он должен реализовывать интерфейс IRichBolt. Проще всего это сделать, унаследовав класс BaseRichBolt. Bolt информирует Storm o результатах своей работы посредством вызова методов класса OutputCollector в методе :

1. – обработка прошла успешно;
2. – обработка завершилась из-за какой-либо ошибки.

В Bolt'ах унаследованных от класса BaseBasicBolt, ack( Tuple tuple ) вызывается после выхода из метода execute автоматически.

При обработке входного Tuple, Bolt может генерировать более одного выходного Tuple. Если Bolt вызвал emit( Tuple sourceTuple,Tuple resultTuple), то образуется направленный ациклический граф (орграф, в котором отсутствуют направленные циклы, то есть пути, начинающиеся и кончающиеся в одной и той же вершине) с вершиной в виде исходного Tuple и потомками в виде порожденных Tuple. Storm отслеживает ошибки процессинга всех узлов графа. В случае возникновения ошибки на любом уровне иерархии, Spout, породивший исходный Tuple, будет уведомлен вызовом fail.

Поскольку Storm является распределенной системой, Tuple могут передаваться с одного узла кластера на другой. В связи с этим Storm обеспечивает отслеживание таймаутов обработки. По умолчанию, весь граф должен быть обработан за 30 секунд, или Storm вызовет метод fail у породившего граф Spout'а. Таймаут можно изменить.

### **Транзакции в Storm**

Представим ситуацию, когда обработка завершилась с ошибкой. Вполне возможно, что потеряна связь с одним из узлов кластера или временно недоступна база данных. В этом случае, нельзя с уверенностью сказать, какие операции выполнились успешно, а какие — нет. Если все операции в цепочке повторно применимы (идемпотентны), например, установка флага, то можно просто перезапустить обработку. Если нет, то на помощь приходят механизмы транзакций Storm.

Когда говорят о характеристиках транзакций, тут же всплывает термин ACID:

1. Atomicity (атомарность). Все изменения, произведенные в системе на протяжении транзакции, либо применяются полностью, либо не применяются совсем;
2. Consistency (cогласованность). Транзакция переводит систему из одного непртиворечивого состояния в другое;
3. Isolation (изолированность). Параллельно выполняемые транзакции не оказывают влияние на результат работы друг друга;
4. Durability (надежность). Зафиксированные транзакцией изменения гарантированно остаются в системе.

В Topology создается объект реализующий интерфейс State, инкапсулирующий работу с БД. Входные данные, поступающие в Spout, разбиваются на Tuple и собираются в пакеты (batch). Batch ассоцируется с уникальным transaction id. Tuple образующие batch могут обрабатываться параллельно.

В конце цепочки обработки, набор Tuple, относящихся к одной транзакции, передается в метод updateState класса, реализующего интерфейс StateUpdater, который и производит модификацию State. В случае успешного завершения, Spout получает уведомление об успехе обработки batch'a. В случае ошибки, Spout должен передать на обработку весь batch повторно.

Таким образом Storm гарантирует, что Batch будет зафиксирован в БД полностью и только один раз.

Storm гарантирует, что Batch'и передаются в StateUpdater строго последовательно, в порядке возрастания transaction id. То есть Batch номер 2 будет зафиксирован только после успешной фиксации Batch'а номер 1.

Транзакционные возможности Storm очень удобно использовать для передачи данных из одной системы в другую, когда требуется нетривиальная обработка. Например, одна система генерирует файлы, Storm их разделяет на записи, обрабатывает в параллельном режиме и складывает в БД. В случае ошибки обработки есть гарантия, что файл не будет удален и не будет обработан дважды.

**EHCACHE**

**2.1 Введение в Ehcache**

Ehcache - основанный на стандартах кеш с открытым исходным кодом. Создан для повышения производительности, разгрузки баз данных и упрощения масштабирования программного продукта. Являясь надежным, проверенным и полнофункциональным решением, он наиболее широко используем на сегодняшний день в приложениях, написанных на Java. Можно использовать Ehcache как кеш общего назначения или как кэш второго уровня для Hibernate. Также можно дополнительно интегрировать его с продуктами сторонних производителей, такими как ColdFusion, Google App Engine и Spring.

Ehcache предоставляет кэш в процессе, который можно реплицировать на нескольких узлах. Он также лежит в основе BigMemory Go и BigMemory Max, коммерческого кэширования Terracotta и хранения данных в памяти. Массив Terracotta Server Array, поставляемый с BigMemory Max, позволяет использовать смешанные конфигурации в процессе или вне процесса с кэшами размером в терабайт.

**2.2 Терминология**

**Cache** - википедия определяет этот термин как “a store of things that will be required in the future, and can be retrieved rapidly.”, что переводится, как хранилище вещей, которые потребуются в будущем, и которые могут быть быстро извлечены. Сам по себе кэш представляет собой набор временных данных, которые либо дублируют данные, расположенные в другом месте, либо являются результатом вычисления. Данные, которые уже находятся в кэше, могут быть повторно доступны с минимальными затратами с точки зрения времени и ресурсов.

**Cache hit** - когда элемент данных запрашивается из кэша и он существует для данного ключа, то это случай носит название “кэш попадание” (cache hit).

**Cache miss** - когда элемент данных запрашивается из кэша и он не существует для данного ключа, то данный случай носит название “кэш промах“ (cache miss).

**System of Record (SOR)** - источник кэшированных данных. Кэш действует как локальное хранилище копий данных, полученных или сохраненных в системе записи (SOR). SOR часто является традиционной базой данных, хотя это может быть специализированная файловая система или какое-либо другое надежное долговременное хранилище. Работая с Ehcache, предполагается, что в качестве SOR используется база данных.

**Standalone cache topology** - автономная кэш топология, при которой данные хранятся в узле приложения. Любые другие узлы приложения независимы, между ними нет связи. Если используется автономная топология, когда несколько узлов используют одно и то же приложение, то между ними существует слабая согласованность. Они содержат согласованные значения для неизменяемых данных или обновляют элемент данных после того, как время жизни элемента завершилось.

**Distributed cache topology** - распределенная кэш топология, данные хранятся на удаленном сервере (или кластере серверов) с поднабором недавно используемых данных, хранящихся в каждом узле приложения. Эта топология предлагает богатый набор параметров для согласованности. Распределенная топология является рекомендуемым подходом в кластерной или масштабируемой прикладной среде. Она обеспечивает высочайший уровень производительности, доступности и масштабируемости. Распределенная топология доступна только с BigMemory Max.

**Replicated cache topology** - реплицированная кэш топология, кэшированный набор данных хранится в каждом узле приложения, и данные копируются или недействительны по узлам без блокировки. Репликация может быть асинхронной или синхронной, где поток записи блокируется во время распространения. Единственный режим согласованности, поддерживаемый в этой топологии, - «Слабая согласованность». Многие производственные приложения развертываются в кластерах из нескольких экземпляров для обеспечения доступности и масштабируемости. Однако без распределенного или реплицированного кэша кластеры приложений демонстрируют ряд нежелательных действий, таких как cache drift и database bottlenecks.

**Cache Drift** - в этом случае, каждый экземпляр приложения хранит свой собственный кеш. Обновления, сделанные в одном кэше, не будут отображаться в других экземплярах. Это происходит с данными веб-сессии. Распределенная или реплицированная топологии кэша обеспечивает синхронизацию всех экземпляров кэша друг с другом.

**Database Bottlenecks** - узкие места в базе данных. В одиночном экземпляре приложения, кэш эффективно защищает базу данных от избыточных запросов. Однако в распределенной среде приложений каждый экземпляр имеет большую нагрузку и обновляет свой собственный кэш. Издержки при загрузке и обновлении нескольких кэшей приводят к узким местам базы данных по мере добавления большего количества экземпляров приложений. Распределенный или реплицированный кэш устраняет накладные расходы для каждого экземпляра загрузки и обновления нескольких кэшей из базы данных.

**Locality of reference** - в информатике - местность ссылки, также известная как принцип локальности, является термином для явления, в котором часто используются одни и те же значения или связанные с ним места хранения, в зависимости от шаблона доступа к памяти. Существует два основных типа эталонного местоположения - временная и пространственная локальность. Временная локальность относится к повторному использованию определенных данных и / или ресурсов в течение относительно небольшой продолжительности времени. Пространственная локальность относится к использованию элементов данных в относительно близких местах хранения. Последовательная локальность - частный случай пространственной локальности - возникает, когда элементы данных упорядочиваются и получают линейный доступ, например, обход элементов в одномерном массиве.

**2.3 Уровни хранения данных**

Вы можете разделить кэш или данные в памяти в следующих областях хранения, называемых уровнями:

1) MemoryStore - память в куче, используемая для хранения элементов кэша. Этот уровень может быть подвержен сборке мусора со стороны Java. Хранилище памяти всегда включено и существует в куче памяти. Оно принимает все данные, будь-то сериализуемые они или нет. Также стоит упомянуть, что это самый быстрый вариант хранения и является потокобезопасным для использования несколькими параллельными потоками. Если вы используете OffHeapStore (доступен только в продуктах BigMemory), MemoryStore хранит копию самого горячего подмножества данных из OffHeapStore.

Все кеши определяют максимальный размер в памяти, с точки зрения количества элементов, во время настройки. Когда элемент добавляется в кеш и выходит за пределы его максимального объема памяти, существующий элемент либо удаляется, если переполнение не активировано, либо оценивается для буферизации на другом уровне, если переполнение включено. Если переполнение включено, выполняется проверка на истечение срока действия хранимых данных. Если срок вышел, то он удаляется, иначе он остается сохраненным, но в отдельном месте.

2) OffHeapStore - предоставляет возможность переполнения памяти MemoryStore. Ограничено по размеру только доступной оперативной памятью. Не подлежит сборке мусора Java (GC). Доступно только для продуктов Terracotta BigMemory. OffHeapStore расширяет кеш к памяти за пределами кучи Java. Этот хранилище, не подлежит воздействию со стороны сборщика мусора Java (GC), ограничено только объемом доступной памяти. Используя OffHeapStore, можно создавать очень большие локальные кеши. OffHeapStore доступен только для продуктов Terracotta BigMemory.

Поскольку данные вне кучи хранятся в байтах, только данные, которые являются Serializable, подходят для OffHeapStore. Любые несериализуемые данные, переполняемые в OffHeapMemoryStore, просто удаляются, и выдается сообщение логгера уровня WARNING. Поскольку сериализация и десериализация происходят при переносе и получении из хранилища не-кучи, теоретически она медленнее, чем MemoryStore. Однако это различие смягчается, когда учитывается сборка мусора, связанная с большими кучами. Для достижения наилучшей производительности рекомендуется выделять для кэша как можно больше памяти кучи, не запуская GC-паузы. Затем используйте OffHeapStore для хранения данных, которые не умещаются в куче (не вызывая GC-паузы).

3) DiskStore - создает резервные копии элементов кэша в памяти и обеспечивает переполнение для других уровней. DiskStore обеспечивает потокобезопасное средство буферизации дисков, которое может быть использовано для дополнительного хранения или сохранения данных при перезагрузке системы. Стоит заметить, что уровень DiskStore доступен только для локальных (автономных) экземпляров кэша. Когда вы используете распределенный кеш (доступный только в BigMemory Max), Terracotta Server Array используется вместо уровня диска. Только данные Serializable могут быть помещены в DiskStore. Записывает и использует диск ObjectInputStream и механизм сериализации Java. Любые несериализуемые данные, переполняющие хранилище на диске, удаляются и генерируется исключение NotSerializableException. Стоить иметь в виду, что на скорость сериализации влияет размер сериализуемых объектов и их тип. Например, было показано, что время сериализации для Java-объекта, состоящего из большого массива Map String, составляет около 126 мс, где сериализованный размер получается 349,225 байт. Также время сериализации для массива байтов составило 7 мс, где сериализованный размер около 310,232 байта. Байт-массивы в 20 раз быстрее сериализовать, что делает их лучшим выбором для повышения производительности дискового хранилища. Конфигурирование хранилища дисков не является обязательным. Если все кэши используют только память и хранилища не-кучи, нет необходимости настраивать хранилище дисков. Это упрощает настройку и использует меньше потоков.

**2.4 Улучшения производительности**

Хоть, Ehcache относится к объектам Java, кэширование используется во всех вычислениях, от кэшей ЦП до системы доменных имен в Интернете (DNS). Это происходит из-за того, что многие компьютерные системы показывают «местность ссылки». Данные, которые находятся рядом с другими данными или недавно использовались, с большей вероятностью будут использоваться снова.



Рисунок 2.4 – Длинный хвост

Крис Андерсон из Wired Magazine придумал термин «длинный хвост» для обозначения случаев в системах электронной коммерции, где небольшое количество товаров может составлять основную часть продаж (или небольшое количество блогов может получить наибольшее количество обращений). И как следствие есть длинный «хвост» менее популярных предметов. Длинный хвост является примером закона силы распределения вероятности, такого как распределение Парето или правило 80:20. Если 20 процентов объектов используется в 80 процентах случаев и можно найти способ снизить затраты на получение этих 20 процентов, производительность системы улучшится.

Стоит задать вопрос о том, на сколько приложение получает преимущество от кэширования. Преимущество особенно заметно, когда есть работа с вводом-выводом и есть нужда в скорости, с которой данные могут быть получены. Если же имеем дело с CPU, тогда время зависит от скорости центрального процессора и основной памяти. Кэширование обычно повышает производительность и снижает нагрузку на веб-сервер.

Приложения с плотной привязкой к CPU, часто ускоряются за счет:

1) улучшения производительности алгоритма;

2) распараллеливания вычислений между несколькими процессорами (SMP) или несколькими машинами (кластерами);

3) увеличения скорости процессора.

Кэш может временно хранить вычисления для повторного использования, включая, но не ограничиваясь большими веб-страницами с высокой стоимостью рендеринга или состояниями аутентификации, когда аутентификация требует криптографических преобразований.

Многие приложения связаны с операциями ввода-вывода, либо дисковыми или сетевыми операциями. В случае баз данных они могут быть ограничены обоими.

Для жестких дисков нет закона Мура. 10 000 оборотов в минуту были быстрыми 10 лет назад и по-прежнему быстры. Жесткие диски ускоряются, используя собственное кэширование блоков в память.

Сетевые операции могут быть замедленны благодаря следующим факторам:

1) время для установки и разрыва соединения;

2) задержка, или минимальное время прохода туда и обратно;

3) пределы пропускной способности;

4) временные затраты на преобразования.

Кэширование является распространенным выходом для повышения производительности приложений, особенно тех, которые постоянно выполняют операции по вводу-выводу. Ehcache очень удобно использовать для кэширования объекта доступа к данным для Hibernate или для кэширования веб-страниц, для страниц, созданных из баз данных.

Следствием повышения производительности является повышенная масштабируемость. Если предположим, что у нас есть база данных, которая может выполнять до 100 дорогих запросов в секунду. Помимо данного порога, база данных резервирует запросы, и если возникают дополнительные соединения, производительность работы с базой данных медленно падает. В этом случае кэширование, вероятно, уменьшит рабочую нагрузку. Если кэширование может привести к тому, что 90% из этих 100 будут хитами кэша и не повлияют на базу данных, база данных может масштабироваться в 10 раз.

**РЕАЛИЗАЦИЯ ПОСТАВЛЕННОЙ ЗАДАЧИ**

В качестве проекта было решено реализовать модуль контроля погодный веб-служб, основанных на протоколе HTTP, использующие REST архитектуру передачи данных. Были разработаны компоненты фреймворка Apache Storm согласно рисунку 2.1. Вместе они работают для сравнения текущих погодных данных с данными на час вперед.

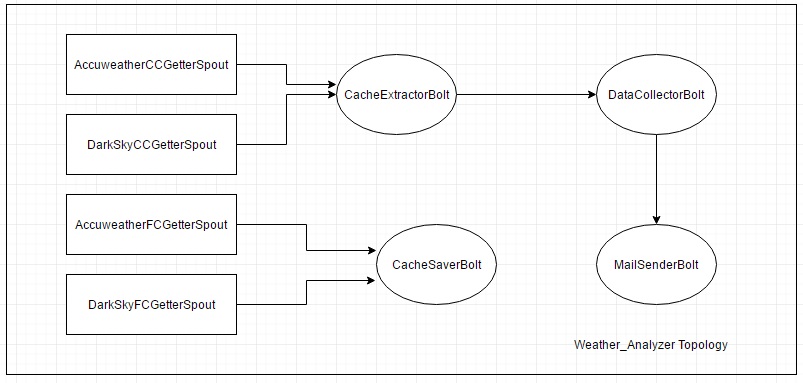


Рисунок 2.1 – Топология реализованного модуля

### **3.1 Получение и сохранение прогноза**

Для получения прогноза на час вперед в данном модуле отвечают DarkSkyFCGetterSpout и AccuweatherFCGetterSpout. Две буквы «FC» посреди являются сокращением «Forecast Condition», что в переводе значит: «Прогноз состояния». Каждый из них обращается к соответствующей веб-службе за погодными данными один раз в час. Получение метеоданных происходит независимо в разных потоках данного модуля.

Веб-службы accuweather.com и darksky.net предоставляет REST API с возможностью делать 1000 запросов в день бесплатно. На рисунке 3.2 и на рисунке 3.3 представлены результаты запросов о метеоданных на час вперед в городе Минске.

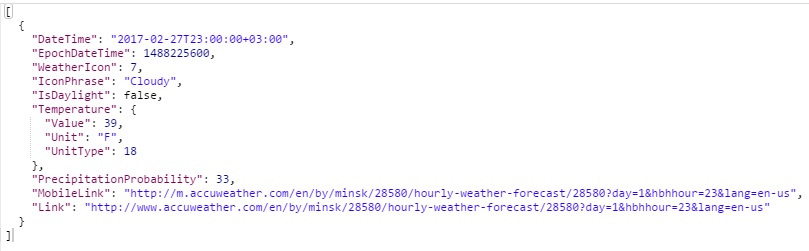


Рисунок 3.2 – прогноз на час от accuweather.com

На рисунке 3.2 представлен пример ответа, в котором находится значение температуры и состояние атмосферы. В ходе дальнейшей разработки данный ответ не будет удовлетворять поставленные требования перед модулем. Сервис accuweather.com показал себя как недостаточно полноценный для решения задач, требующих детализированный прогноз.

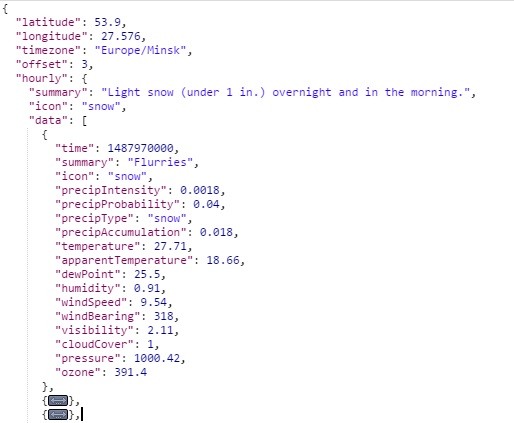


Рисунок 3.3 – прогноз от darksky.net

На рисунке 3.3 представлен пример ответа от darksky.net. В нем наблюдается детализированный прогноз на час вперед. В ответе присутствуют данные о скорости ветра, состоянии атмосферы, количестве озона, давлении. Работа с darksky.net имеет перспективы в дальнейшей разработке погодного модуля.

Как можно заметить, каждая веб-служба имеет свою модель данных, поэтому было принято решение о создании собственной, общей для всех структуры, с которой будет проще и удобнее взаимодействовать в дальнейшем. Общая модель представлена на рисунке 2.4.

Рисунок 3.4 – модель объекта общего типа

В объекте WeatherConditionTO последние две буквы являются сокращением «Transfer Object», что в переводе значит: «объект передачи». В дальнейшем у данной сущности появятся новые свойства, такие как значение давления, состояния атмосферы.

После получения данных о прогнозе и преобразовании их к объекту модели общего вида, идет передача их в CacheSaverBolt. На этом этапе поступившему объекту создается ключ, затем он целиком помещается в кэш.

**3.2 Получение текущего и извлечение сохраненного состояния**

Для получения текущего состояния погоды используются AccuweatherCCGetterSpout и DarkSkyCCGetterSpout. Две буквы «СС» являются сокращением «Current Condition», что в переводе значит: «текущее состояние». Обращение к веб-службам происходит один раз в час. На рисунке 2.5 и рисунке 2.6 представлены результаты запросов о метеоданных на текущее состояние в городе Минск.



Рисунок 3.5 – текущее состояние от accuweather.com

Было упомянуто выше, что ответ от accuweather.com не содержит достаточно деталей для дальнейшей работы. Рисунок 2.5 подтверждает это еще один раз.

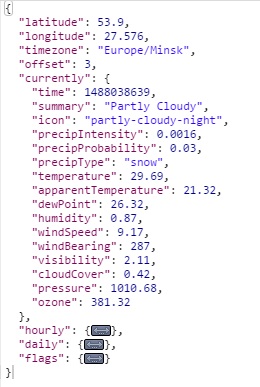


Рисунок 3.6 – текущее состояние от darksky.net

Рисунок 3.6 также, как и рисунок 3.3 доказывает, что веб-служба darksky.net способна довольно точно и полно описать как прогноз, так и текущее состояние

После успешного преобразования в объект общей модели, идет его передача в CacheExtractorBolt, где происходит поиск и извлечения из кэша прогноза, который был помещен туда ровно час назад. Если объект будет найден, то произойдет передача данных о прогнозе и о текущем состоянии в DataCollectorBolt для дальнейшего анализа и генерации отчета.

**3.3 Обработка и отчет**

В случае, когда в DataCollectorBolt пришли два соответствующих объекта модели WeatherConditionTO, создается оценка погрешности для данного города от соответствующей веб-службы. Оценка сохраняется в кэш в виде объекта LocationAccuracy, модель которого изображена на рисунке 3.7. Затем формируется объект для будущего включения в отчет, его модель изображена на рисунке 3.8.

Рисунок 3.7 – модель объекта оценки точности прогноза за час

Данный объект будет храниться с целью дальнейшего его извлечения для выбора наилучшей веб-службы для определенного населенного пункта. В качестве идентификатора пункта используется поле locationKey.

Рисунок 3.8 – модель объекта для генерации отчета

Для создания отчетности и отправки ее на почту, указанную в настройках отвечает MailSenderBolt. Один раз в час он достает объекты типа HourAccuracy и создает таблицу в формате csv, которую в дальнейшем можно будет открыть с помощью Microsoft Excel. Данный файл будет отправлен на почту ответственному лицу. Пример содержимого файла csv изображено на рисунке 3.9.

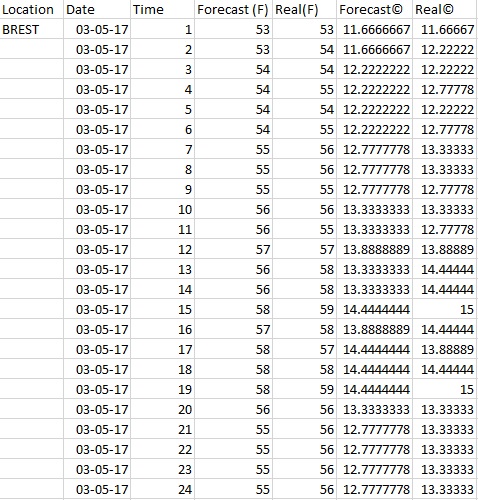


Рисунок 3.9.1 – пример отчета по погоде, отправленный на почту

В представленном рисунке 3.9.1 прослеживается погрешность. Именно она и определяет лидера среди поставщиков метеоданных. Нетрудно вывести диаграмму, согласно рисунку 3.9.1– рисунок 3.9.2

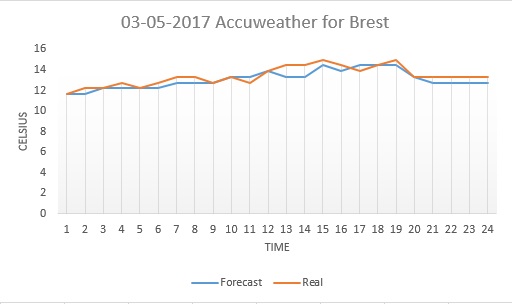


Рисунок 3.9.2 – диаграмма по рисунку 3.9.1

На рисунке 3.9.2 видна диаграмма, которая четко показывает в какой промежуток дня погрешность высока. В данном случае это в период с 13 часов дня до 16 вечера. Данные диаграммы позволят в дальнейшем за короткий срок понять где именно имеется большая погрешность, и на какую веб-службу стоит обратить особое внимание.

**ЗАКЛЮЧЕНИЕ**

В данной работе:

* изучен фреймворк Apache Storm и взаимодействие его компонентов;
* изучена работа с отправкой писем на электронные почтовые ящики;
* рассмотрен спектр погодных веб-служб, предоставляющих бесплатный доступ к метеоданным;
* разработан высоконагруженный модуль получения, обработки, и предоставления отчета о данных, полученных от метеослужб.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Эккель Б. Философия Java. Библиотека программиста. – СПб.: Питер, 2001. – 880 с.: ил;
2. Буч Г. Объектно-ориентированный анализ и проектирование. 2-е изд. Пер. с англ. – М.: «Издательство Бином», 1998 – 560с.: ил;
3. Хабибуллин И. Ш. Создание распределённых приложений на Java 2. – СПб.: БХВ-Петербург, 2001 г. – 688с.: ил;
4. Java. Промышленное программирование : практ. пособие / И.Н. Блинов, В.С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с;
5. Шаблоны проектирования в Java / М. Гранд; Пер. с англ. С. Беликовой. – М.: Новое знание, 2004. – 559 с.: ил;
6. Материалы ресурса Habrahabr [Электрон. ресурс] Habrahabr Project – Режим доступа: https://habrahabr.ru/ – Дата доступа: 16.02.2017;
7. Материалы свободной энциклопедии Wikipedia [Электрон. ресурс] / Wikipedia Project – Режим доступа: http://wikipedia.org – Дата доступа: 02.02.2017.