Министерство образования Республики Беларусь

Учреждение образования
Белорусский государственный университет информатики и радиоэлектроники

Факультет компьютерных систем и сетей

Кафедра информатики

Отчет по лабораторной работе
по курсу «Проектирование программных систем обработки больших объемов информации»
на тему «**Обработка данных о вакансиях**»

Выполнил студент группы 956241: Зязюлькин С.П.

Проверил: Стержанов М.В.

Минск, 2021

Постановка задачи

Задача заключается в доработке лабораторной работы «Обработка данных о вакансиях» из предыдущего семестра:

1. Добавить параллельную обработку данных о вакансиях.
2. Добавить интеграционную шину, реализовать поступление данных из разных источников.
3. Реализовать хранение данных в NoSQL БД.
4. Развернуть приложение в облаке.
5. Покрыть созданный код тестами.
6. Провести анализ предметной среды и обосновать выбор решения.

План выполнения работы

1. Изучить существующие облачные решения для обработки больших объемов информации.
2. Добавить интеграционную шину.
3. Добавить NoSQL БД.
4. Добавить параллельную обработку данных.
5. Реализовать поступление и обработку данных.
6. Написать тесты.
7. Провести анализ предметной среды и обосновать выбор решения.
8. Сделать выводы.

Облачные решения для обработки больших объемов информации

В настоящее время одними из самых популярных облачных решений для обработки больших объемов информации являются Amazon Web Services, Microsoft Azure и Google Cloud Platform.

Amazon Web Services – облачная платформа Amazon, созданная в 2006 году. Стала первооткрывателем в данной области. AWS предоставляет более 70 услуг с широким спектром покрытия по всему миру. Серверы доступны в 14 географических регионах.

Microsoft Azure представляет собой многогранную сложную систему, которая обеспечивает поддержку множества различных услуг, языков

программирования и фреймворков. Система была запущена в 2010 году и развивается очень быстрыми темпами. В составе облака более 60 служб и центров обработки данных в 38 различных географических регионах.

Google Cloud Platform является самой молодой облачной платформой и, в первую очередь, удовлетворяет потребности поиска Google и Youtube. В настоящее время у компании представлено более 50 услуг и 6 глобальных центров обработки данных.

Для решения лабораторной работы было выбрано облачное решение Microsoft Azure.

Интеграционная шина

Служебная шина Microsoft Azure – это полностью управляемый брокер сообщений корпоративного типа с поддержкой очередей сообщений и разделов публикации и подписки. Служебная шина используется для разделения приложений и служб, что предоставляет следующие преимущества:

- распределение нагрузки между конкурирующими рабочими ролями;
- безопасная маршрутизация для передачи данных и команд управления через границы служб и приложений;
- координация транзакционных работ, которые требуют высокой надежности.

Очереди сообщений позволяют хранить сообщения, пока принимающее приложение сможет получить и обработать их. Сообщения в очередях упорядочиваются и получают метку времени поступления. Принятое брокером сообщение всегда надежно сохраняется в хранилище с тройной избыточностью, распределенном между зонами доступности (если для пространства имен включена эта возможность). Служебная шина не сохраняет в памяти или временном хранилище сообщения, о приеме которых клиенту уже отправлено подтверждение. Сообщения доставляются в режиме запроса (после получения запроса). В отличие от модели с опросом занятости, которая реализована в некоторых других облачных очередях, операция извлечения здесь может существовать долго, вплоть до появления доступного сообщения.

Рисунок 1 – Очередь сообщений служебной шины

В рамках данной лабораторной работы будет использоваться очередь сообщений для хранения новых данных о вакансиях. Данные в эту очередь могут поступать параллельно из разных источников.

Первым делом необходимо создать саму служебную шину. Обзор созданной шины представлен на следующем рисунке.

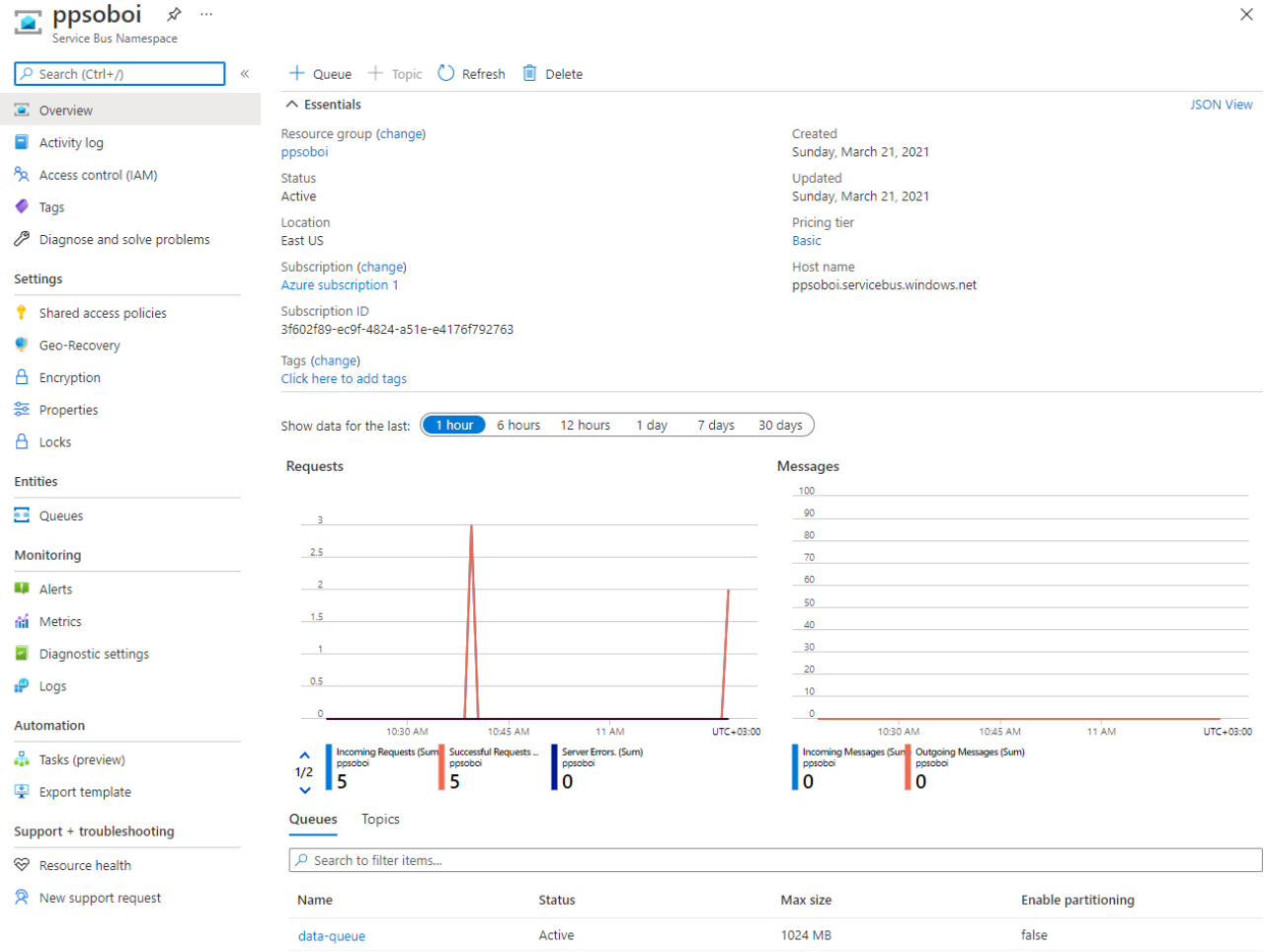

Рисунок 2 – Служебная шина Microsoft Azure

После создания шины необходимо создать очередь сообщений, с которой будут работать внешние источники.

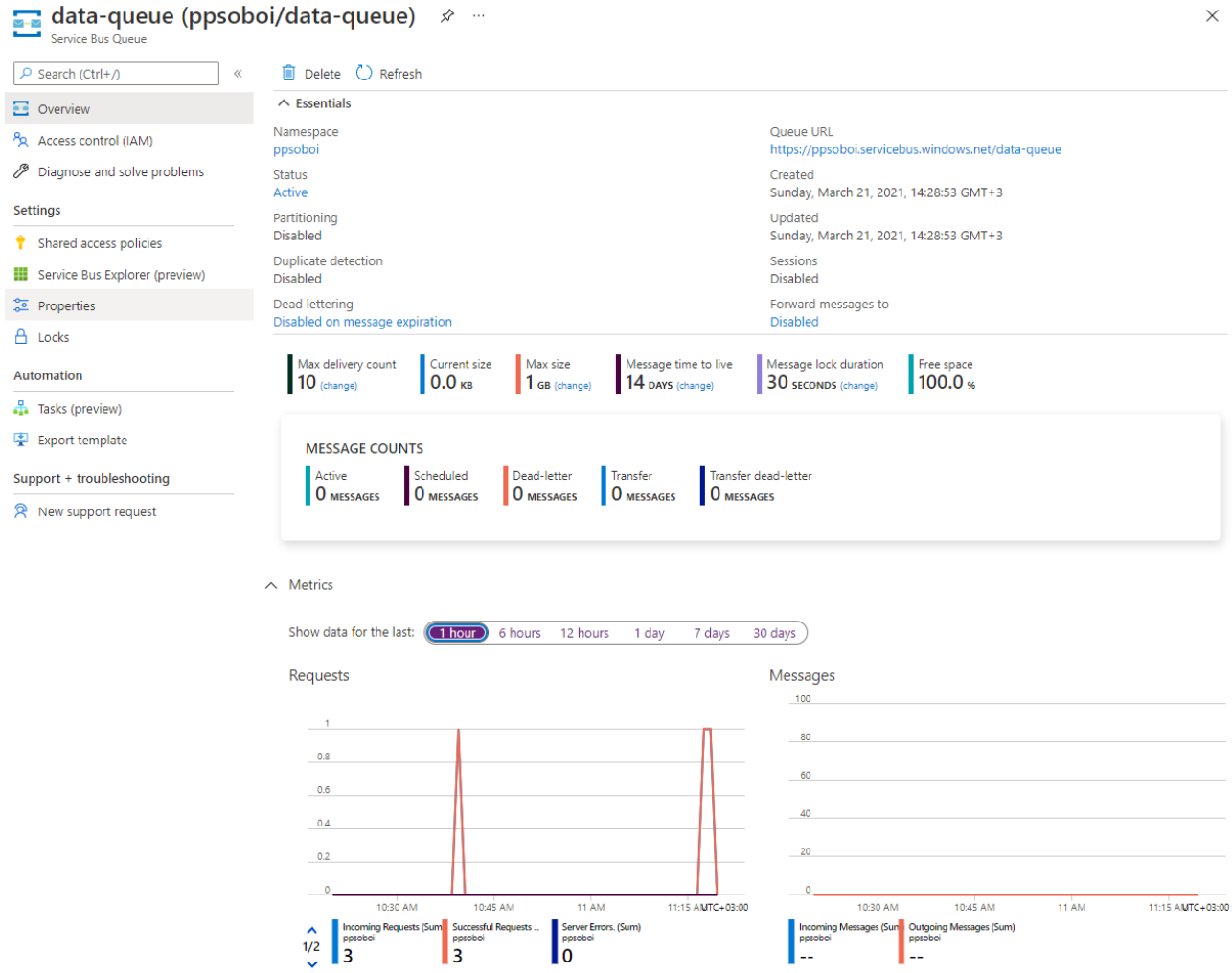

Рисунок 3 – Очередь сообщений Microsoft Azure

NoSQL БД

Azure Cosmos DB – это глобально распределенная, многомодельная служба базы данных Microsoft, необходимая для работы с критически важными приложениями. Эта служба базы данных обеспечивает глобальное распределение, гибкое масштабирование пропускной способности и хранилища по всему миру, задержки менее 10 миллисекунд на уровне 99-го процентиля, а также гарантированную высокую доступность – все это согласно ведущим в отрасли соглашениям об уровне обслуживания. Azure Cosmos DB автоматически индексирует данные без необходимости управлять схемой и индексом. Так как эта база данных является многомодельной, она поддерживает модели данных документа, «ключ – значение», графа и столбчатые модели данных. Служба Azure Cosmos DB реализует сетевые протоколы для стандартных API NoSQL, включая Cassandra, MongoDB, Gremlin

и Хранилище таблиц Azure. Это позволяет использовать привычные клиентские драйверы и средства NoSQL для взаимодействия с базой данных Cosmos DB.

Для использования в лабораторной работе было выбрано API MongoDB.

Рисунок 4 – Взаимодействие с Azure Cosmos DB

В Cosmos DB будут храниться предобработанные данные вакансий. Эти данные будут накапливаться со временем, а также использоваться для выполнения анализа собранных данных.

Для создания базы данных необходимо сначала создать учётную запись Azure Cosmos DB.

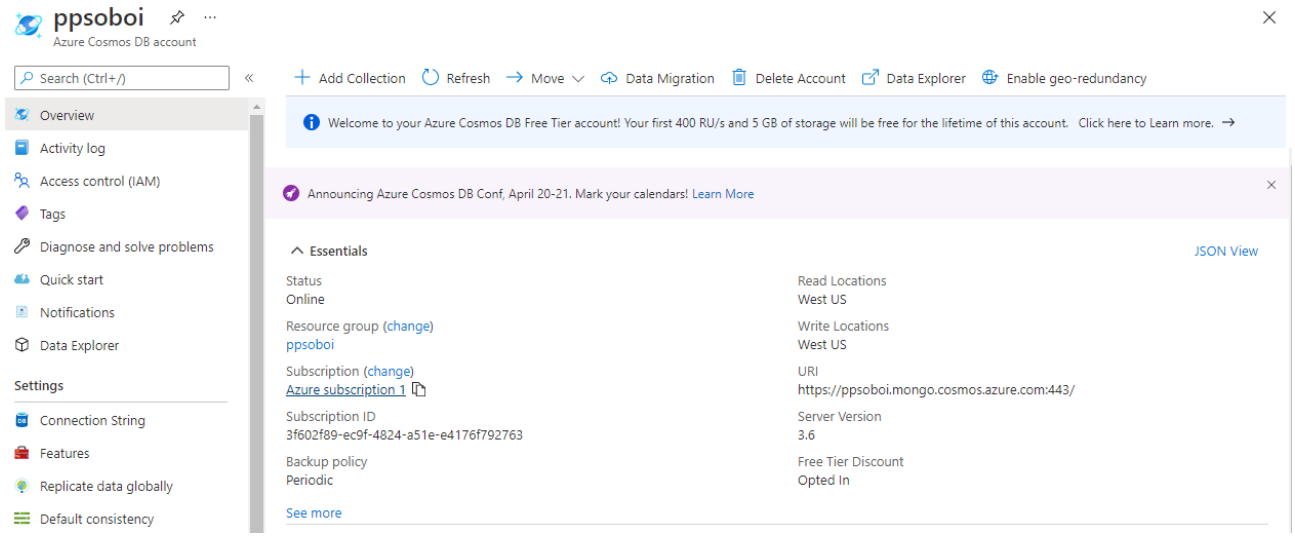

Рисунок 5 – Учётная запись Azure Cosmos DB

После создания учётной записи необходимо создать базу данных и коллекцию в ней.

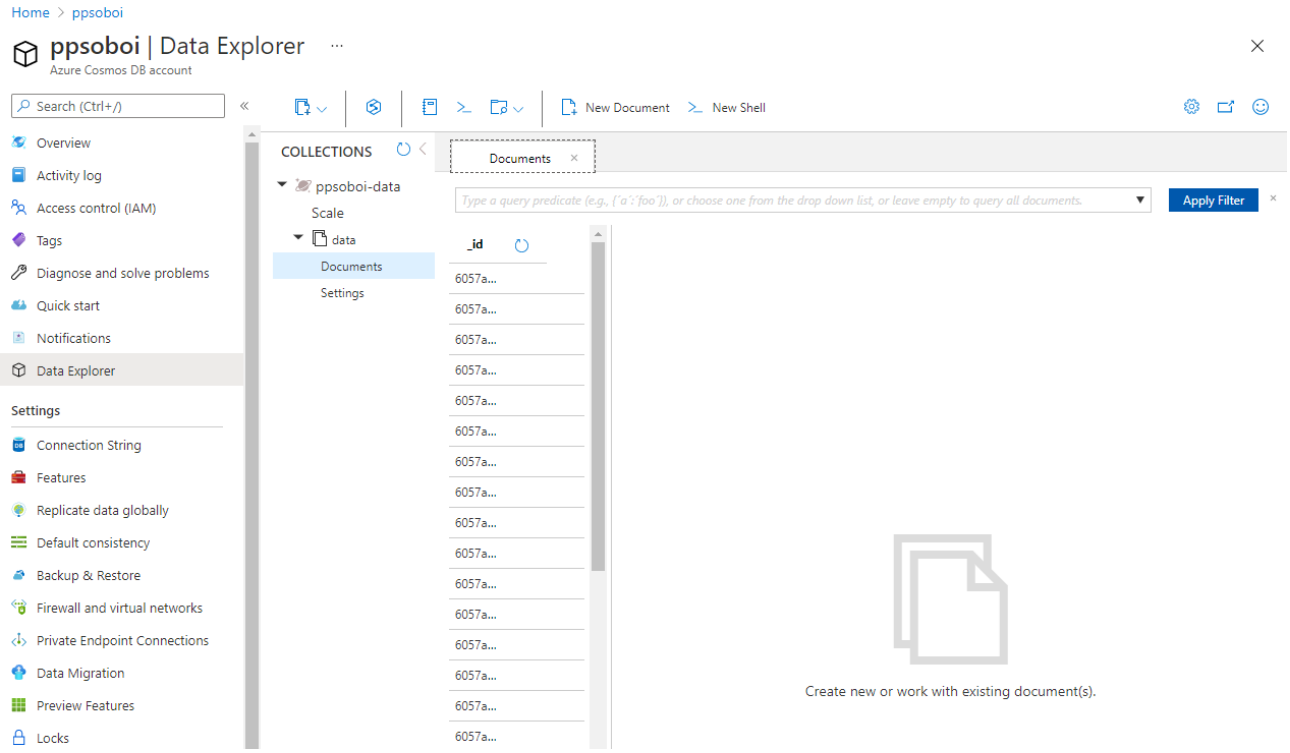

Рисунок 6 – База данных ppsoboi-data с единственной коллекцией data

Параллельная обработка данных

Azure Functions – это serverless решение, позволяющее писать меньше кода и упрощающее поддержку инфраструктуры. Azure Functions позволяет группировать логику приложения в блоки, называемые функциями. Функции могут реагировать на возникновение разного рода событий. Microsoft Azure позволяет скалировать ресурсы, требуемые для работы функций, предоставляя дополнительные ресурсы и создавая дополнительные инстансы функций в зависимости от текущей нагрузки.

В лабораторной работе будут использоваться две функции. Первая (data-preprocessing) отвечает за обработку поступающих в очередь новых данных о вакансиях и сохранение данных в Cosmos DB. Вторая функция (analysis) отвечает за извлечение данных из Cosmos DB и их анализ.

Azure Functions реализовывались с использованием ЯП Java и среды разработки IntelliJ IDEA. Основной код функций приведён ниже.

```
@FunctionName("data-preprocessing")
public void run(
        @ServiceBusQueueTrigger(
                name = "message", queueName = "data-queue", connection =
"MyStorageConnectionAppSetting")
                RawJobView message,
```

```
        final ExecutionContext context
) {
    context.getLogger().info(String.format("Processing job '%s'...",
message.getName()));

    try {
        final ProcessedJobView processedJob = new ProcessedJobView();
        processedJob.setName(parseName(message.getName()));
        processedJob.setMinSalary(parseMinSalary(message.getSalary()));
        processedJob.setMaxSalary(parseMaxSalary(message.getSalary()));
        processedJob.setPlace(parsePlace(message.getPlace()));
        processedJob.setEmployer(parseEmployer(message.getEmployer()));

processedJob.setDescription(parseDescription(message.getDescription()));

        final MongoDatabase mongoDatabase = mongoClient.getDatabase(DB_NAME);
        final MongoCollection<Document> collection =
mongoDatabase.getCollection(COLLECTION_NAME);
        collection.insertOne(processedJob.toDocument());

        context.getLogger().info(String.format("Processed job '%s'",
message.getName()));
    } catch (final Exception e) {
        context.getLogger().info("Failed to process job: " + e.getMessage());
    }
}


@FunctionName("analysis")
public HttpResponseMessage run(
        @HttpTrigger(
            name = "req",
            methods = {HttpMethod.POST},
            authLevel = AuthorizationLevel.ANONYMOUS)
            HttpRequestMessage<Optional<Request>> request,
        final ExecutionContext context) {

    final Request req = request.getBody().orElse(null);
    context.getLogger().info(String.format("Processing request %s...", req));

    if (req == null) {
        context.getLogger().info("Processed request null: - ignored");
        return request.createResponseBuilder(HttpStatus.BAD_REQUEST).body("null
request").build();
    } else if (!req.isValid()) {
        context.getLogger().info(String.format("Processed request %s: invalid
request - field is null", req));
        return
request.createResponseBuilder(HttpStatus.BAD_REQUEST).body("invalid
request").build();
    } else {
        final MongoDatabase mongoDatabase = mongoClient.getDatabase(DB_NAME);
        final MongoCollection<Document> collection =
mongoDatabase.getCollection(COLLECTION_NAME);

        final Bson group = Aggregates.group("$" + req.field,
Accumulators.sum("count", 1));
```

```
        final Bson project = Aggregates.project(Projections.fields(
                Projections.excludeId(), Projections.include("count"),
Projections.computed("value", "$_id")));

        final Bson sort = Aggregates.sort(Sorts.descending("count"));
        final Bson limit = Aggregates.limit(req.getLimit() != null ?
req.getLimit() : 10);

        final List<Document> results =
                collection.aggregate(Arrays.asList(group, project, sort,
limit)).into(new ArrayList<>());

        final String response =
results.stream().map(Document::toJson).collect(Collectors.joining(","));
        context.getLogger().info(String.format("Processed request %s: OK",
req));
        return request.createResponseBuilder(HttpStatus.OK).body("[" + response
+ "]").build();
    }
}
```

Созданные функции представлены на следующем рисунке.

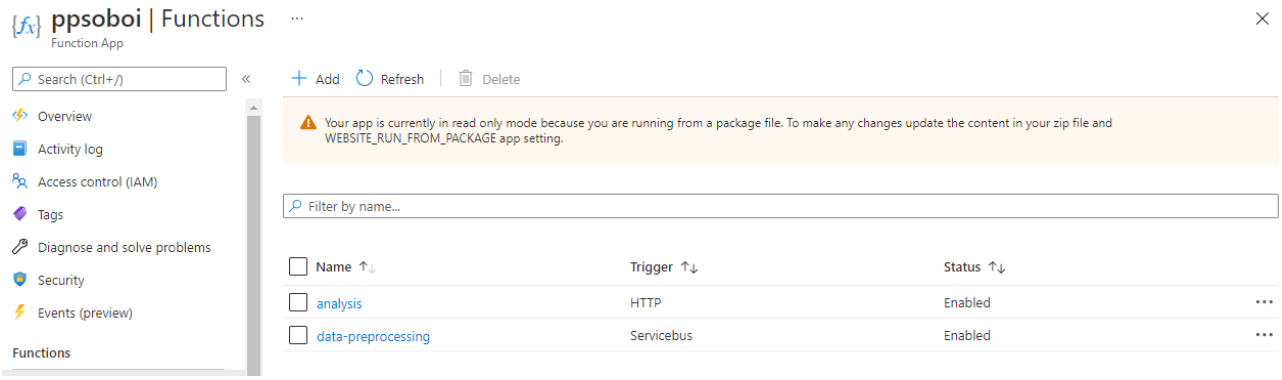

Рисунок 7 – Azure Functions: data-preprocessing и analysis

Поступление и обработка данных

Выгрузка данных делается с сайтов, агрегирующих вакансии. Примером такого сайта является https://russia.trud.com. Выгрузка данных осуществлялась с использованием ЯП Python и фреймворка Scrapy.

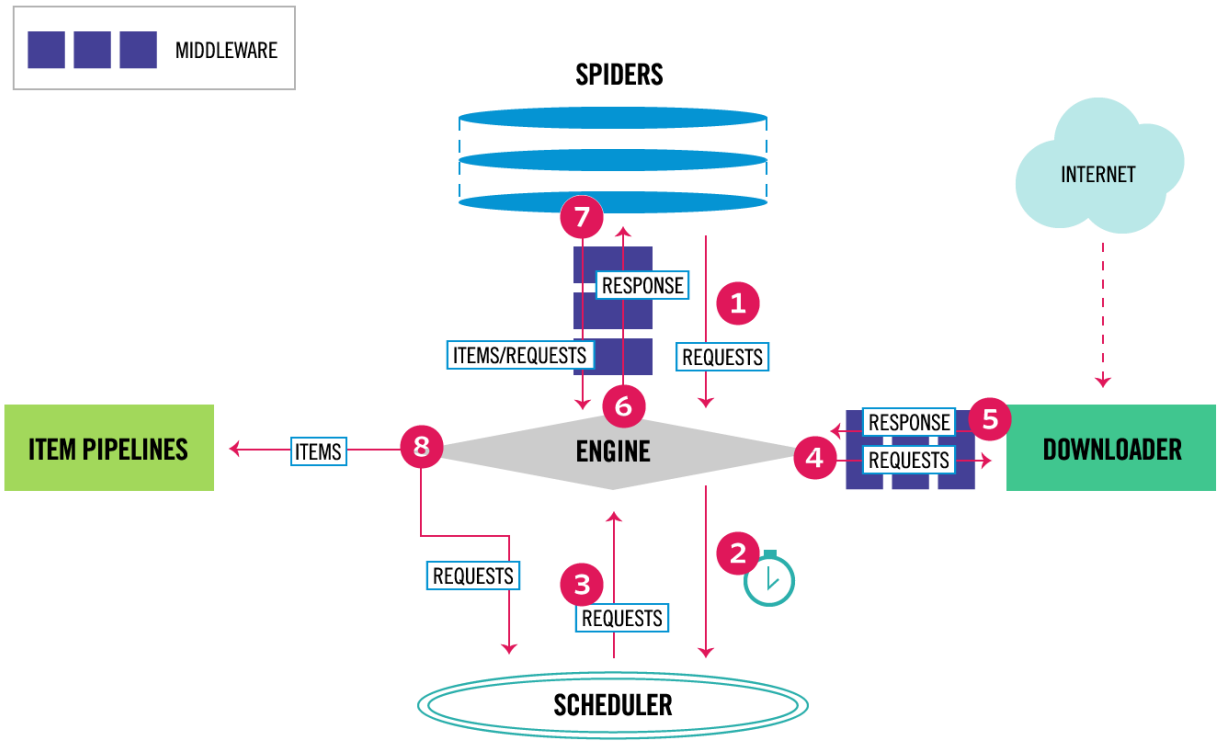

Рисунок 8 – Архитектура фреймворка Scrapy

Для загрузки данных в очередь служебной шины использовался отдельный скрипт на ЯП Python и библиотека azure-servicebus. Основной код этого скрипта представлен ниже.

```
servicebus_client =
ServiceBusClient.from_connection_string(conn_str=CONNECTION_STR,
logging_enable=True)

with servicebus_client:
    sender = servicebus_client.get_queue_sender(queue_name=QUEUE_NAME)

    with sender:
        for i, job_json in enumerate(jobs_json):
            print(f'sending message {i}...')
            message = ServiceBusMessage(json.dumps(job_json))
            sender.send_messages(message)
```

Загрузка данных о вакансиях может осуществляться параллельно с разных сайтов.

После попадания в очередь служебной шины данные обрабатываются функцией data-preprocessing, после чего отправляются в Cosmos DB. Часть лога функции представлена ниже.

```
2021-03-21T20:37:27.501 [Information] Processing job 'Кок'...
2021-03-21T20:37:27.549 [Information] Processed job 'Кок'
2021-03-21T20:37:27.549 [Information] Function "data-preprocessing" (Id:
76dd986f-fbf0-4ef5-ac04-22d93beea8a6) invoked by Java Worker
2021-03-21T20:37:27.549 [Information] Executed 'Functions.data-preprocessing'
(Succeeded, Id=76dd986f-fbf0-4ef5-ac04-22d93beea8a6, Duration=49ms)
2021-03-21T20:37:27.753 [Information] Executing 'Functions.data-preprocessing'
(Reason='New ServiceBus message detected on 'data-queue'.', Id=d5d6ba74-da8d-
4c12-9a29-59cd30125b03)
2021-03-21T20:37:27.753 [Information] Trigger Details: MessageId: 9c0c2531-
859c-4c0f-82b7-d30a25f914d2, DeliveryCount: 1, EnqueuedTime: 3/21/2021 8:37:27
PM, LockedUntil: 3/21/2021 8:37:57 PM, SessionId: (null)
2021-03-21T20:37:27.755 [Information] Processing job 'Фельдшер'...
2021-03-21T20:37:27.805 [Information] Processed job 'Фельдшер'
2021-03-21T20:37:27.805 [Information] Function "data-preprocessing" (Id:
d5d6ba74-da8d-4c12-9a29-59cd30125b03) invoked by Java Worker
2021-03-21T20:37:27.805 [Information] Executed 'Functions.data-preprocessing'
(Succeeded, Id=d5d6ba74-da8d-4c12-9a29-59cd30125b03, Duration=52ms)
2021-03-21T20:37:28.038 [Information] Executing 'Functions.data-preprocessing'
(Reason='New ServiceBus message detected on 'data-queue'.', Id=6ef9b03f-6bc7-
4e52-b6ef-18ffcdb4b069)
2021-03-21T20:37:28.038 [Information] Trigger Details: MessageId: c81f45b8-
01d2-4389-bfd2-d750992cf83a, DeliveryCount: 1, EnqueuedTime: 3/21/2021 8:37:27
PM, LockedUntil: 3/21/2021 8:37:58 PM, SessionId: (null)
2021-03-21T20:37:28.039 [Information] Processing job 'Снайпер'...
2021-03-21T20:37:28.090 [Information] Processed job 'Снайпер'
2021-03-21T20:37:28.091 [Information] Function "data-preprocessing" (Id:
6ef9b03f-6bc7-4e52-b6ef-18ffcdb4b069) invoked by Java Worker
2021-03-21T20:37:28.091 [Information] Executed 'Functions.data-preprocessing'
(Succeeded, Id=6ef9b03f-6bc7-4e52-b6ef-18ffcdb4b069, Duration=53ms)
2021-03-21T20:37:28.181 [Information] Executing 'Functions.data-preprocessing'
(Reason='New ServiceBus message detected on 'data-queue'.', Id=d96c1745-4082-
47b6-81d2-1c8f9c72ebf3)
2021-03-21T20:37:28.181 [Information] Trigger Details: MessageId: 4f176a83-
56f3-4982-ac92-5866764164a5, DeliveryCount: 1, EnqueuedTime: 3/21/2021 8:37:28
PM, LockedUntil: 3/21/2021 8:37:58 PM, SessionId: (null)
```

Данные в Cosmos DB можно просматривать в веб-интерфейсе Microsoft Azure.

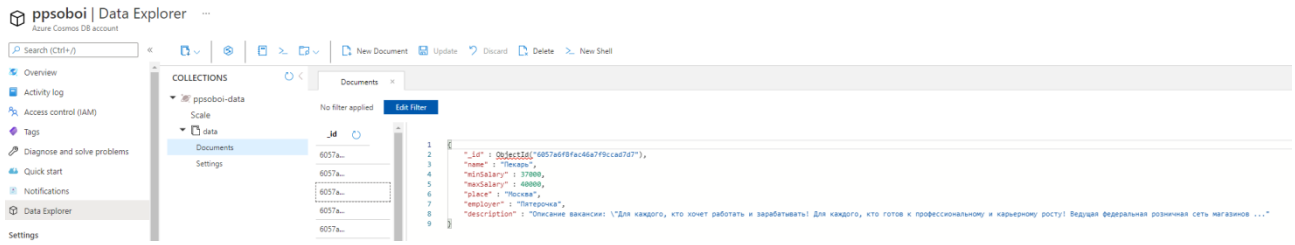

Рисунок 9 – Просмотр данных Cosmos DB

Также в веб-интерфейсе можно просмотреть данные мониторинга шины.

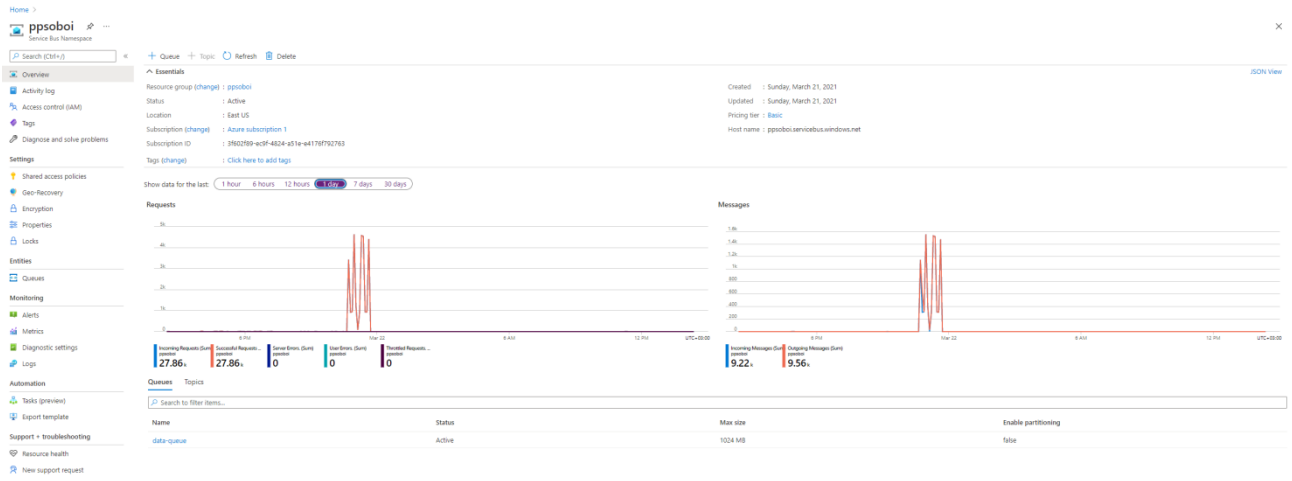

Рисунок 10 – Мониторинг служебной шины

Для обработки загруженных данных используется функция analysis. Эта функция обрабатывает http-запрос, собирает аналитику по заданным запросом требованиям, возвращает в ответе на запрос.

Пример поиска трёх наиболее популярных вакансий представлен на рисунке ниже.

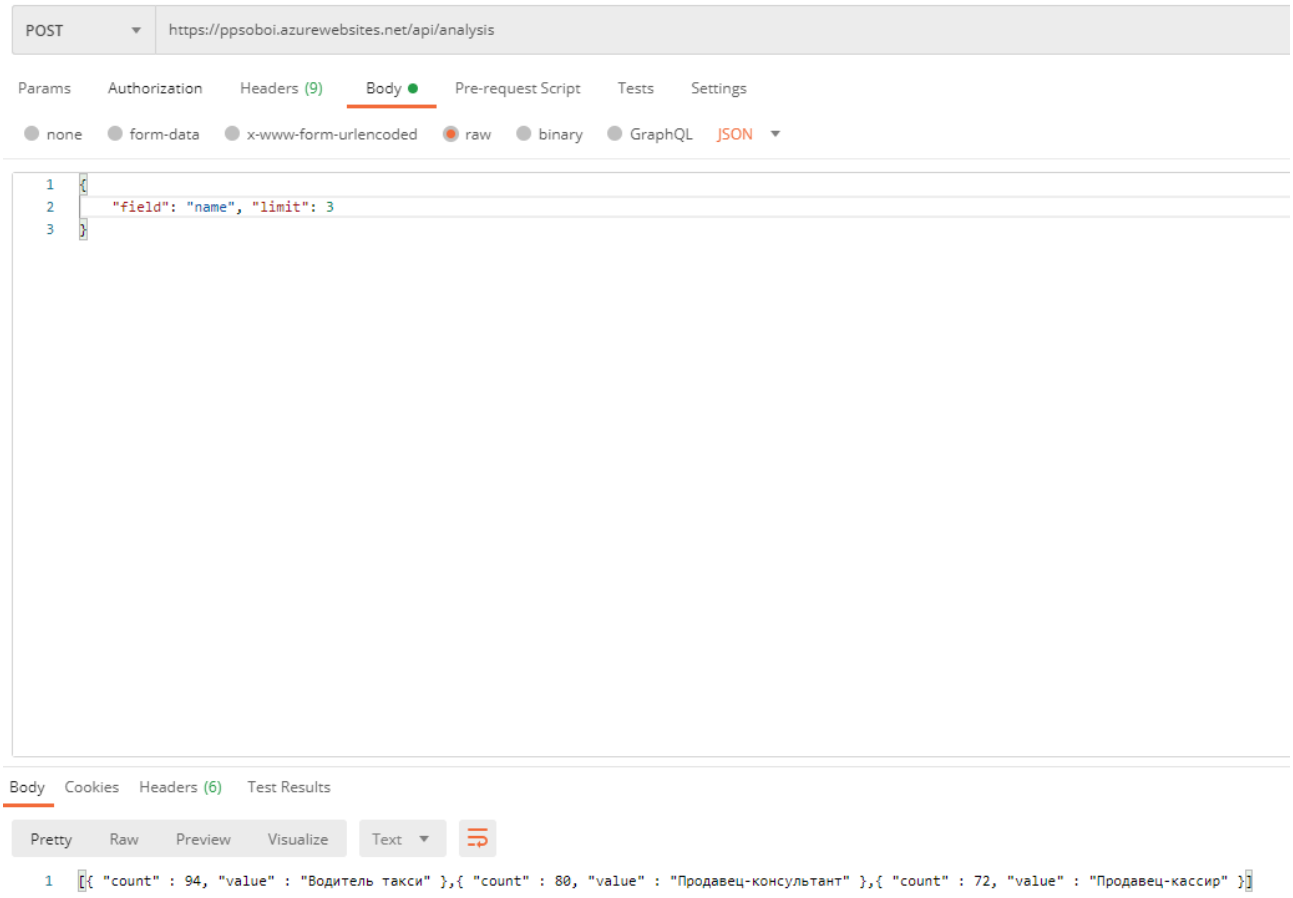

Рисунок 11 – Анализ собранных данных

Итоговая обработка данных выполнялась в Jupyter Notebook на ЯП Python.

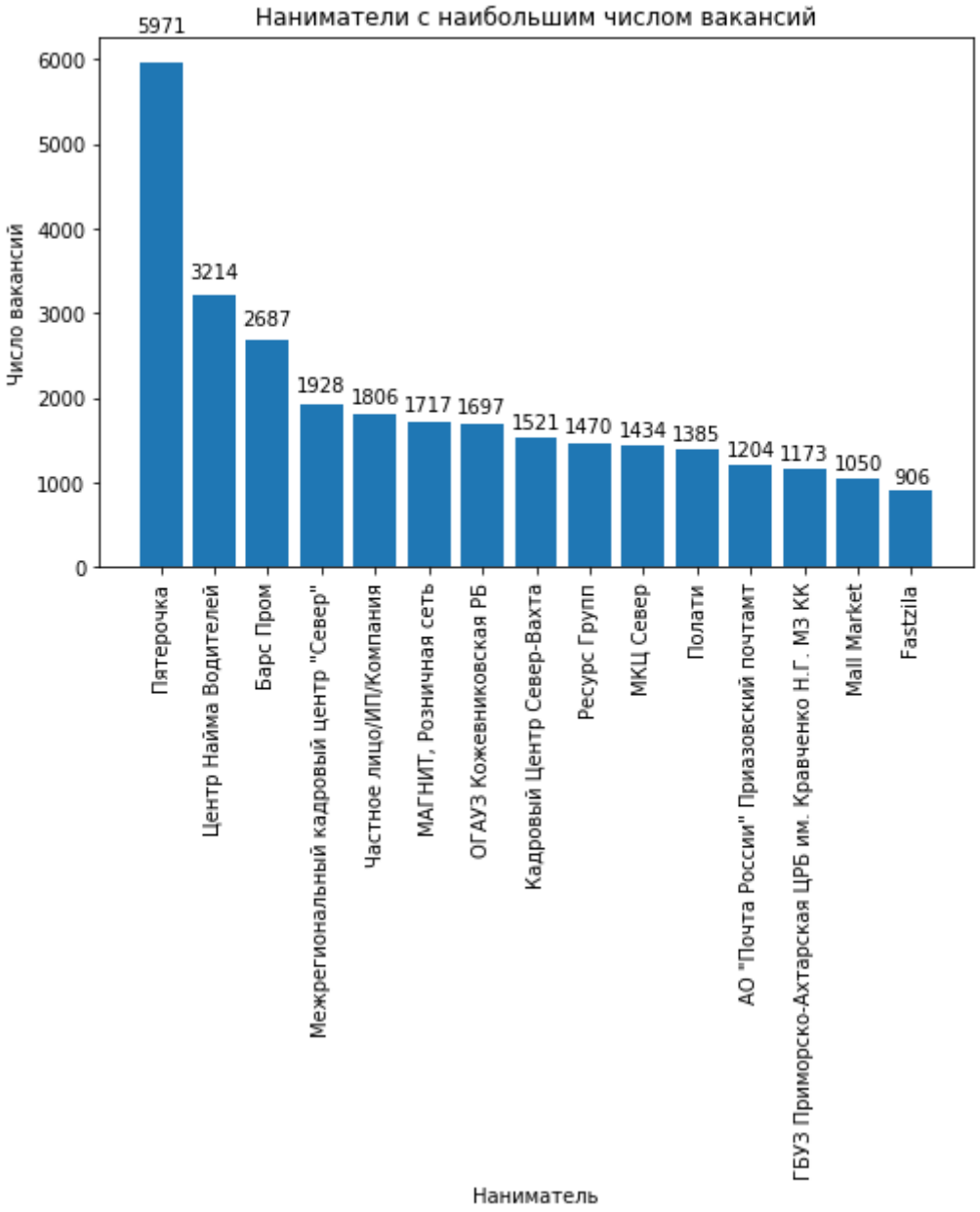

Рисунок 12 – Наниматели с наибольшим числом вакансий

Тесты

Код лабораторной работы написан на двух ЯП: Python, Java.

Для тестирования кода, написанного на Python, используется модуль unittest. Этот модуль поддерживает автоматизацию тестов, использование

общего кода для настройки и завершения тестов, объединение тестов в группы, а также позволяет отделять тесты от фреймворка для вывода информации.

Код одного из тестов, проверяющего отправку batch-а сообщений в сервисную шину, представлен ниже.

```
def test_sending_batch_message(self):
    with self.servicebus_client:
        sender = self.servicebus_client.get_queue_sender(queue_name=QUEUE_NAME)
        with sender:
            batch_message = sender.create_message_batch()
            for _ in range(5):
                batch_message.add_message(ServiceBusMessage("{}"))
            sender.send_messages(batch_message)
```

Для тестирования кода, написанного на Java, используется библиотека JUnit. JUnit – библиотека для модульного тестирования программ Java. Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений – Jmock, EasyMock, DbUnit, HttpUnit и т. д.

Также для написания тестов на Java понадобился фреймворк Mockito. При тестировании кода (прежде всего юнит-тестировании, но не только) тестируемому элементу часто требуется предоставить экземпляры классов, которыми он должен пользоваться при работе. При этом часто они не должны быть полнофункциональными – наоборот, от них требуется вести себя жёстко заданным образом, так, чтобы их поведение было простым и полностью предсказуемым. Такие объекты называются заглушками (stub). Чтобы их получить, можно создавать альтернативные тестовые реализации интерфейсов, наследовать нужные классы с переопределением функционала и так далее, но всё это достаточно неудобно, избыточно и чревато ошибками. Более удобное во всех смыслах решение – специализированные фреймворки для создания заглушек. Одним из таковых и является Mockito. В частности, при написании тестов понадобилось добавить заглушки для входных и выходных объектов Azure Functions.

Код одного из тестов, проверяющего корректность обработки запроса аналитики при помощи Azure Functions, представлен ниже.

```
@Test
public void testCorrectRequest() {
    final HttpResponseMessage correctResponse =
Mockito.mock(HttpResponseMessage.class);
    final HttpResponseMessage incorrectResponse =
Mockito.mock(HttpResponseMessage.class);
```

```
    final HttpResponseMessage.Builder correctResponseBuilder =
Mockito.mock(HttpResponseMessage.Builder.class);

Mockito.doReturn(correctResponseBuilder).when(correctResponseBuilder).body(Mock
ito.anyString());
    Mockito.doReturn(correctResponse).when(correctResponseBuilder).build();

    final HttpResponseMessage.Builder incorrectResponseBuilder =
Mockito.mock(HttpResponseMessage.Builder.class);

Mockito.doReturn(incorrectResponseBuilder).when(incorrectResponseBuilder).body(
Mockito.anyString());
    Mockito.doReturn(incorrectResponse).when(incorrectResponseBuilder).build();

    final HttpRequestMessage<Optional<HttpTriggerFunction.Request>> request =
            Mockito.mock(HttpRequestMessage.class);

    final HttpTriggerFunction.Request requestData = new
HttpTriggerFunction.Request("name", 2);
    Mockito.doReturn(Optional.of(requestData)).when(request).getBody();

Mockito.doReturn(incorrectResponseBuilder).when(request).createResponseBuilder(
Mockito.any(HttpStatus.class));

Mockito.doReturn(correctResponseBuilder).when(request).createResponseBuilder(Ht
tpStatus.OK);

    final ExecutionContext context = Mockito.mock(ExecutionContext.class);
    Mockito.doReturn(Logger.getGlobal()).when(context).getLogger();

    final HttpResponseMessage response = new HttpTriggerFunction().run(request,
context);
    Assert.assertSame(correctResponse, response);
}
```

Все написанные тесты успешно проходят.

Анализ предметной среды

Любой веб-ресурс становится ценным и посещаемым только при наличии полезного, интересного пользователям и уникального контента. «Кто владеет информацией, тот владеет миром».

Сегодня объемы информации превосходят возможности их обработки у любого даже самого талантливого человека или узкопрофильного специалиста. Поэтому для автоматического сбора и обработки больших объемов информации был придуман скрапинг (он же – парсинг) веб-сайтов.

В широком понимании веб-скрапинг – это сбор данных с различных интернет-ресурсов. Общий принцип его работы можно объяснить следующим

образом: некий автоматизированный код выполняет GET- или POST-запросы на целевой сайт и, получая ответ, парсит HTML-документ, ищет данные и преобразует их в заданный формат.

Существует масса решений для скрапинга веб-сайтов. Например, проекты с открытым кодом на разных языках программирования: Goose, Scrapy – Python; Goutte – PHP; Readability, Morph – Ruby. В рамках данной лабораторной работы было выбрано решение Scrapy из-за его простоты и удобства ЯП Python.

При скрапинге веб-сайтов возникает необходимость обработки большого объёма полученных данных.

Большие данные (англ. big data) – серия подходов, инструментов и методов обработки структурированных и неструктурированных данных огромных объемов и значительного многообразия для получения воспринимаемых человеком результатов, эффективных в условиях непрерывного прироста, распределения по многочисленным узлам вычислительной сети, сформировавшихся в конце 2000-х годов, альтернативных традиционным системам управления базами данных и решениям класса Business Intelligence.

Основные принципы работы с такими данными:

- Горизонтальная масштабируемость. Поскольку данных может быть сколь угодно много – любая система, которая подразумевает обработку больших данных, должна быть расширяемой. В 2 раза вырос объём данных – в 2 раза увеличили количество железа в кластере и все продолжило работать.
- Отказоустойчивость. Принцип горизонтальной масштабируемости подразумевает, что машин в кластере может быть много. Это означает, что часть этих машин будет гарантированно выходить из строя. Методы работы с большими данными должны учитывать возможность таких сбоев и переживать их без каких-либо значимых последствий.
- Локальность данных. В больших распределённых системах данные распределены по большому количеству машин. Если данные физически находятся на одном сервере, а обрабатываются на другом – расходы на передачу данных могут превысить расходы на саму обработку. Поэтому одним из важнейших принципов проектирования BigData-решений является принцип локальности данных – по возможности обрабатывать данные на той же машине, на которой они хранятся.

С лавинообразным ростом информации в мире и необходимости ее обрабатывать за разумное время встала проблема вертикальной масштабируемости баз данных – рост частот процессора сильно замедлился, скорость чтения с диска также растет медленными темпами, плюс цена мощного сервера всегда больше суммарной цены нескольких простых серверов. В этой ситуации обычные реляционные базы, даже кластеризованные на

массиве дисков, не способны решить проблему скорости, масштабируемости и пропускной способности. Единственный выход из ситуации – горизонтальное масштабирование, когда несколько независимых серверов соединяются быстрой сетью и каждый владеет/обрабатывает только часть данных и/или только часть запросов на чтение-обновление. В такой архитектуре для повышения мощности хранилища (емкости, времени отклика, пропускной способности) необходимо лишь добавить новый сервер в кластер – и все. Это стало одной из причин развития NoSQL баз данных. Процедурами шардинга, репликации, обеспечением отказоустойчивости (результат будет получен, даже если один или несколько серверов перестали отвечать), перераспределения данных в случае добавления ноды занимается сама NoSQL база.

Существуют готовые облачные решения для обработки больших объемов данных: Amazon Web Services, Microsoft Azure, Google Cloud Platform.

Для решения лабораторной работы был выбран Microsoft Azure. В частности, Azure Service Bus, Azure Functions и Cosmos DB. Причины выбора:

- Новизна технологии. Мне не доводилось ранее работать с Microsoft Azure, в отличие от, например, Amazon Web Services.
- Microsoft Azure имеет выгодные бесплатные предложения. Так, например, в первые 12 месяцев доступны 750 часов и 13 миллионов операций в служебной шине, 400 RU/s с 25 GB места в Azure Cosmos DB. Также постоянно на бесплатной основе доступен миллион вызовов Azure Functions в месяц.

Выводы

В рамках выполнения лабораторной работы было проделано следующее:

1. Добавлена параллельная обработка данных о вакансиях при помощи Azure Functions.
2. Добавлена интеграционная шина Azure Service Bus, реализовано поступление данных в шину.
3. Реализовано хранение данных в NoSQL БД – Cosmos DB.
4. Развернуто приложение в Microsoft Azure.
5. Созданный код покрыт тестами.
6. Проведен анализ предметной среды.

В рамках выполнения лабораторной работы:

1. Изучил веб-интерфейс облачной платформы Microsoft Azure.
2. Научился работать с Azure Service Bus, загружать данные в шину при помощи библиотеки azure-servicebus на ЯП Python.

3. Научился работать с Azure Functions, разрабатывать Azure Functions на ЯП Java.

4. Изучил API MongoDB и научился работать с Cosmos DB на ЯП Java.

5. Развил навыки обработки данных в Jupyter Notebook на ЯП Python.

6. Развил навыки написания тестов с использование модуля unittest на ЯП Python.

7. Развил навыки написания тестов с использованием библиотеки JUnit на ЯП Java.