Distributed Property Statistics

Jack Geraghty(16384181) - Thomas Creavin (17311103) - Wyman Cheung (17330316)

December 2020

1 Project Overview

Project Description

This project aims to create a proof-of-concept of a highly-scalable, fault-tolerant distributed system for computing statistics on large data sets. For the proof-of-concept, the team opted to design the system to provide property statistics based on data collected from Irish property sites like Daft.ie and MyHomes.ie. The goal of this system is to allow web users to get statistics on sample property data. For this proof-of-concept, the project is most concerned with creating a distributed, fault-tolerant back end. The data used is sample data, the queries are limited, and the interface is plain; these caveats can be improved with time but it is important to ensure the back end is well-designed.

Motivation

There are many people interested in property statistics: home-buyers, investors, statisticians, and journalists. It is difficult to gather specific property statistics; there isn't a nationwide database for property data, and even if there was, the queries that can be performed are likely limited. These stakeholders could benefit from a website that can quickly and easily compute custom statistics on property data scraped from various sources.

Design Consideration

In order for the system to be capable of processing a large volume of requests, the system must be distributed: the web client must be scale to handle many uses; the requests must be distributed to different database pullers to query the database quickly; the database must be fault-tolerant, scalable, and performant to handle all the queries; the queries and data must transferred to distributed statistics efficiently and in fault-tolerant manner; and the statistics must be computed scalably.

It's evident from these constraints that the system must be distributed, scalable, and fault-tolerant to be successful. Two added benefits of this approach are individual services can be scaled horizontally as needed and the solution can be made generic to any type of metric by inserting different service.

2 Technologies

Amazon DynamoDB

Amazon DynamoDB is a key-value and document database that delivers single-digit millisecond performance at any scale. It's a fully managed, multi-region, multi-active, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications.

The data preparation module (the 'Puller') leverages DynamoDB to store historical property data at scale. Since property data is unstructured and the project requires a horizontally scalable database, the team opted to use a NoSQL database over an SQL database. Specifically, this project uses DynamoDB over other NoSQL databases like MongoDB because it is fully managed, highly performant, and easy to use.

Pros

- Easy to scale
- Fault-tolerant
- Fully managed
- Highly performant
- Low cost

Cons

- By default, DynamoDB is available and partition tolerant but not consistent; it's eventually-write-consistent.
- By default, it guarantees consistency and durability but not atomicity and isolation.
- Data queries are limited compared to SQL.
- Must be deployed on AWS.

Rest

REST API, or RESTful API is an architectural style for distributed systems. There are some constraints that define a RESTful system - Uniform interface, Client-Server Architecture, Statelessness.

Uniform Interface simplifies and decouples the architecture to allow each part to evolve independently. Client-Server constraints are the separating of user interface concerns. Statelessness is the extrinsic state stored on each client which consists of information dependent on a server's context.

In our project REST is used to send resources from the Client to the Load Balancer, and the Load Balancer to the Pullers. The Load Balancer and Puller both implement the Spring @RestController annotations in order to expose the endpoints necessary for resource transfer. The resource, in this case, is the RequestMessage. This is a message which contains the query populated by the homepage of the client.

While the Client uses the standard Spring Controller for the user to access the user interface, it sends the queries via Rest to the Load Balancer. A major difference between a standard Controller and RestController is that a Response Body does not need to be included in the latter. This simplifies the job of the Load Balancer and Puller.

Pros

- Easy to scale
- Easy to understand
- Highly flexible
- Lower latency
- High level of concurrency

Cons

- Cannot maintain states in REST
- Lack of security means that it is only suitable for public URLS.

Kafka

Apache Kafka is an open-source **stream processing** software built by the Apache Foundation. Kafka aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. It is starting to become more widely used in recent years with many large tech companies adopting the technology for large-scale distributed systems.

In our project, Kafka is the backbone of the system in terms of information transfer, as well as enabling easy scaling for any of the components consuming from our Kafka topics through Kafka's consumer group

concept. Consumer groups group together consumers which all consume from the same Kafka topic. When this is combined with topic partitions, concurrent consumption from topics is made simple. Topic partitions essentially split a topic into some arbitrary number of partitions which can then be consumed from concurrently. Provided that the data is published to topic partitions in a way that doesn't break data consistency then efficient multiprocessing of the data can be performed.

In Distributed Property Statistics, any action starts with a *RequestMessage* being sent from a client. This message contains a query that will be used to supply the Statistics Processors with data eventually. Through REST, an instance of a *Puller* will receive a *RequestMessage*, gather all the information related to the query(i.e. all the matching property data), and then publishes this data to the Kafka topic, *Requests*, in the form of a *BatchMessage* containing all the individual *PropertyMessage*. In this section of Kafka, it will publish in a round-robin way to the partitions in the *Requests* topic.

The number of partitions in the *Requests* topics is set at startup through an *env* file and through *docker-compose*. In the Distributed Property Statistics system, the number of partitions will be set to be the same as the number of *statistics processor* instances that will be run. This ensures that each instance gets its own partition to consume fro. Due to a *BatchMessage* containing all the information relating to a single query then data consistency is ensured when arriving at a *statistics processor* instance. Kafka imposes a condition on the number of partitions and consumers consuming from these partitions. The number of consumers cannot be greater than the number of partitions available as if all consumers consume from a single partition each then there will be partitions not being consumed from and thus data will be left waiting.

Once a *statistics processor* is finished processing a batch of messages it needs to be able to ensure that the client who made the request will receive the result. This is done through the *Results* topic. This topic is split into partitions equal to the number of client instances running. Then each client instance on startup is allocated a single partition by Kafka automatically. This partition allocation is just an integer number, which can be used to publish to that specific partition. The partition ID is included in all *RequestMessages* sent to the services, then at the end of processing the *statistics processor* has access to this ID and can publish back to the correct topic.

Pros

- Easy to scale
- Persistent, data can be stored locally and then loaded on startup
- High throughput
- Low latency
- High level of concurrency

Cons

- Requires dedicated resources, it runs independently of other services
- Lack of monitoring tools makes it difficult to inspect what is happening

Kafka was easy to integrate into the project and provided a simple way of communicating between services. It's functionality enables the scaling of most of the services and provides excellent means of ensuring that data is not lost in the event of failure. While in a production system it does require dedicated resources which has it's operation costs it does provide more resilience to failure, this is achieved by managing a Kafka cluster across multiple hosts so that if one fails the rest won't since they're on a separate machines. It does lack dedicated monitoring tools but it does offer a selection of useful command line scripts for consuming and producing from topics.

Docker Integration

Each module of this project is built into a Docker container allowing for easy deployment to different machines. Docker allows the passing of environment variables making it easy to configure each component. All of the services use the same network to enable communication between the different services.

3 Implementation

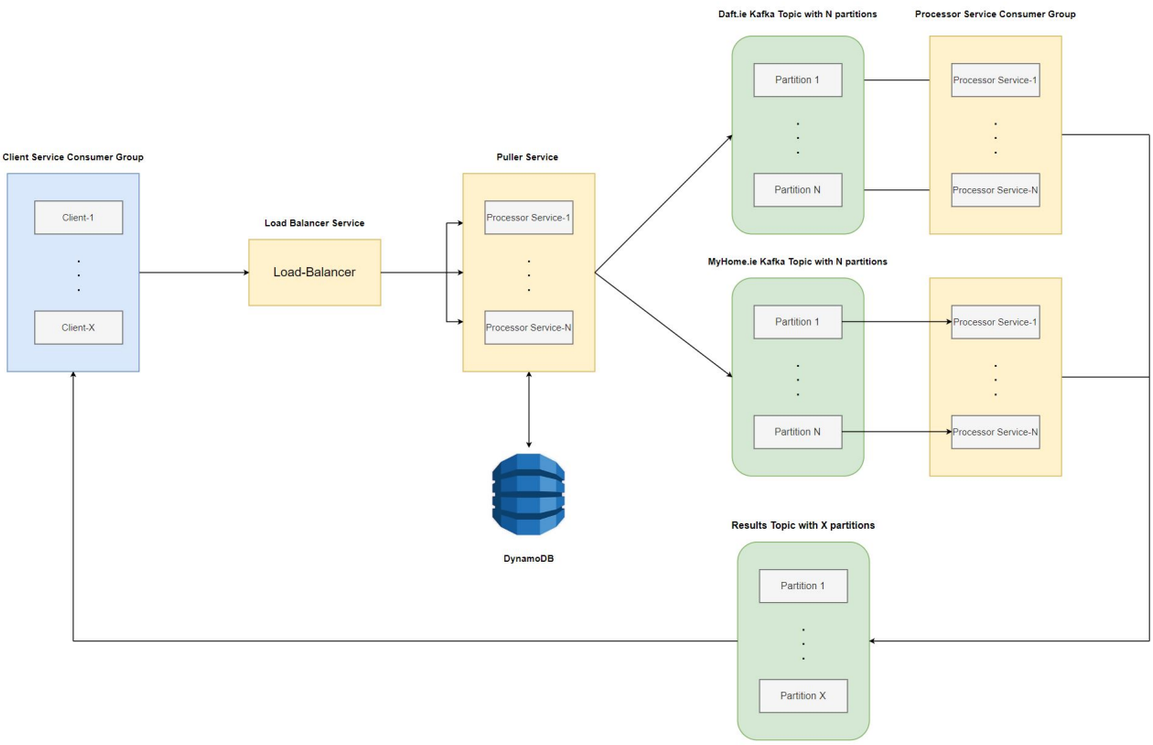

Figure 1: Architecture of Distributed Property Statistics

3.1 Client

The Client's task is to provide users a UI to check property statistics. The Client also sends the RequestMessage to the load balancer for distribution. The client comprises of 2 main components: *ClientController* and *ResultsHandler*.

ClientController is the main component in which the endpoints are created. It posts the queries from the homepage to the load balancing endpoint. It also processes the results and displays them with google visualisation charts once results are polled.

ResultsHandler handles the initialisation of the KafKaConsumer. It contains a run method to continuously poll for results and inputs the results into a map.

Load Balancer

The Load Balancers task is to distribute the load evenly amongst the available Pullers. This is done via round-robin in which each Puller's workload is shared evenly.

3.2 Puller

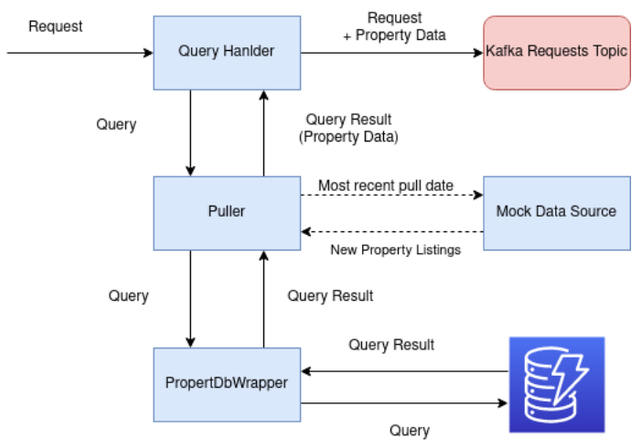

Figure 2: Simplified overview of the Puller architecture

The Puller is the project's data preparation module. For every property query request, the Puller polls a historical property database for relevant property data; the property request and the relevant data are

published to Kafka to be processed. The Puller keeps the historical database up-to-date by regularly pulling in new property data from data sources.

The puller comprises of four main components: *QueryHandler*, *Puller*, *MockDataSource*, and *PropertyDbWrapper*.

QueryHandler is the main component of this module. QueryHandling is a three-step process: first, a query is received from the client; second, the handler asks the Puller class for data relating to the query; and last, the handler packages the query request and query data together and publishes it to a Kafka requests topic.

Puller is responsible for querying the database and keeping the database up-to-date. It accepts query requests from the $QueryHanlers$, queries DynamoDB using the $PropertyDbWrapper$, and returns the query results. Additionally, the $Puller$ uses a scheduled thread to check the $MockDataSource$ for new property listings and writes them to DynamoDB.

MockDataSource emulates a remote property data source. In a real system, this component could be a data scrapper that would routinely scrape Daft.ie's or MyHome's API, or this component could be a connection to an existing property database.

This component creates sample property data for apartments and houses for all thirty-two counties. To have somewhat realistic pricing data, the generated prices use the current average price for that county.

PropertyDbWrapper is responsible for interfacing with DynamoDB, the property database. All the DynamoDB API calls are made using this wrapper. The benefits of using a wrapper class are the methods that can be tested more easily and for teammates unfamiliar with DynamoDB, it's easier for them to use the wrapper.

The wrapper configures DynamoDB as so: each data source e.g Daft.ie is given a table; the property listing's unique ID serves as the partition key and its listing date serves as the sort key; each property table has a global secondary index comprising of the county and listing date; queries are performed on the global secondary index.

3.3 Statistics Processor

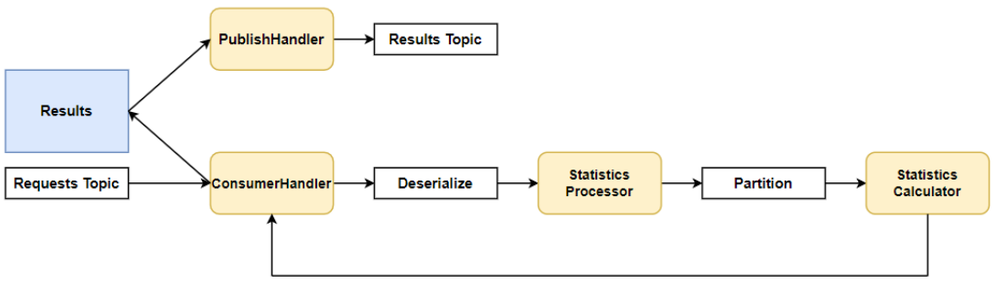

Figure 3: Overview of the Statistics Processor Architecture

The statistics processor is responsible for producing the statistics for a given set of properties. It is designed to be very flexible and scalable. A topic name is the only thing required by an instance of the *statistics processor*. It then joins the consumer group of that topic and consumes *BatchMessages* to produce results. By having it only rely on a topic name it enables the service to be highly scalable. This is however based on the assumption that data being published into the topic is self-contained, that is all data relating to the query is contained in that *BatchMessage*, and the *statistics processor* will be able to complete a request using only that information.

The *statistics processor* service is split into two main components, the ***ConsumerHandler*** and the ***PublishHandler***. Both of these are then used through composition, to create the ***MesssageHandler***. The role of the *MessageHandler* is to coordinate the two handlers and allow for shared access to a *ConcurrentHashMap* which stores the results of processing. On startup, a *MessageHandler* is created and then the *ConsumerHandler* and the *PublishHandler* get started in their own thread.

The *ConsumerHandler* is made up of a **KafkaConsumer**, for consuming messages from a Kafka topic, a **MessageDeserializer**, which is used for deserializing the messages consumed from Kafka and then an **ExecutorService**, which is used to start new threads for each batch received. When a message is consumed from Kafka, it is then deserialized and the necessary information extracted. This information is the *partitionID*, the messages UUID and then the *BatchMessage* to be processed. A new thread is then started and a *StatisticsProcessor* calculates the range of statistics provided. The results of this are then put into the

results map of the *MessageHandler*.

The ***StatisticsProcessor*** is used to partition each of the *PropertyMessages* received in a batch into a map based on the query provided by the user. This is done through a custom *TemplateBuilder* and *Partitioner* class. This level of partitioning handles dividing the data based on the broad categories of *County*, **PropertyType** and *PostcodePrefix*. An example of such a partition is given below.

```
{
    "county", "Mayo",
    "propertyType", "house"
    "postcodePrefix", null
    .
    .
    .
}
Resulting partition: "Mayo_house"
```

Each of these partitions is then used a Map key, whose value is the *PropertyMessages* that match that partition. Each entry in this partitioned map is then sent to be processed by a *StatisticsCalculator* object.

The ***StatisticsCalculator*** object performs another level of partitioning, this time by the date associated with the *PropertyMessage*. Then using an online method(see Welford's algorithm) the standard deviation, variance, and mean are calculated for each of the resulting partitions. A *StatisticsResult* is then returned to the *StatisticsProcessor* and in turn, the collection of *StatisticsResults* for each partition is returned to the *ConsumerHandler* to be put into the results map.

The **PublishHandler** then periodically checks to see if there are any entries in the results map, and if so publishes those results to the correct partition in the *Results* topic. To do this publishing the *PublishHandler* consists of a *MessageSerializer* and a *KafkaProducer*.

4 Scaling

All components with the DSP architecture were designed with scaling in mind and as a result, each component can be easily scaled when starting the service or by adding new hosts when the service is already running.

Docker has been used to deploy each of the services and docker-compose has been used to provide environment variables as well as start each service. Once an image has been created for a service docker-compose is then used to launch the services with the appropriate environment variables. Each of the components is designed to require as little information as possible to work and have as few dependencies as possible on the other services.

Starting with the client, which is implemented using Spring. An instance of the client is responsible for sending requests via REST to the load balancer which then distributes the requests to instances of the puller service. It is possible to scale the client as required since it only needs to be able to connect to a load balancer instance. The only information needs to scale the client is then knowledge of which load balancer to connect to, which can be specified on startup. This is a theoretical problem that could be encountered if the system was receiving a very large number of requests. DSP doesn't fully address this problem in its current state but it would be straightforward to avoid. Currently, a client sends a request to a constant URL for the load balancer instance. A solution to this problem would be to pass in the URL of a specific load balancer on startup, then client instances can be put into a group that corresponds to specific load balancer URLs, thus mitigating the request throughput problem.

Another scaling issue occurs when wanting to add a new client instance to the system. By setting the results topic to have more partitions than clients each client consumer will be assigned more than more partition. This doesn't present an issue in terms of ensuring a result is returned to the correct client instance as when publishing a result from the Statistics module it is published to the first partition assigned to the client. This means that any extra partitions aren't receiving any data but they do turn out to be very useful. A new client instance can be started without taking the entire system offline as Kafka will re-balance the topic partitions amongst the client instances and these re-assigned partitions will then start receiving results provided the new clients make a request.

Kafka enables easy scaling of the Statistics service. The Statistics service is only dependent on the Kafka address and the topic to consume from. Both of which are provided at startup and don't change thereafter, even in the case of Kafka failing(more on this in the Fault Tolerance section). Multiple instances of the statistics service can be started to form a consumer group for the given Kafka topic. Each instance will be assigned a number of partitions in a topic and will consume from them in a round-robin fashion. If a new instance of the Statistics service is added then Kafka will re-balance the partitions to match the new number of consumers. The information needed to then publish the results(the partitionID of the client making the

request) of processing is contained within the message and operates without further knowledge. To add a new instance of the Statistics service one just needs to start up a new instance on a host.

The simplest service is the Load Balancing service which handles balancing requests amongst instances of the Puller service. Changes are required to enable the scaling of this service but it is a straightforward change that wasn't done due to time constraints. Multiple Load Balancer services can be started, each with its own unique IP address. Then the Client service would be divided into groups, each of which would be associated with an instance of the Load Balancer service. The IP of the Load Balancer instance would be provided to each Client instance in a group at startup. The same would then be done on the Puller service side. This method of scaling raises a Fault Tolerance issue which would be if an instance of the Load Balancer fails, this is discussed below in the following section.

The Puller service can be scaled horizontally by deploying more Puller instances and increasing the database provisioning. The puller database is the primary bottleneck. Since this project is using DynamoDB, the property tables can be scaled easily by provisioning more read or write capacity. DynamoDB is equipped with AWS auto-scaling, so CloudWatch can be used to manage the scaling of DynamoDB.

In this project, each data source has its own table. If the team wanted to further optimize the database, each data source could be given its own database; the data source's data could be partitioned across many tables where each table represents a year (or another regular period) of property data; this would allow DyanmaoDB to highly provision the most popular tables and lowly provision the others, and the tables would be smaller making it easier to query.

5 Fault Tolerance

Fault Tolerance was an integral design consideration for DSP. There are several key points in the architecture that are prone to faults and as such when designing these components their ability to handle faults was either included in the solution or was thought about and would be possible to include provided there was more time. In the following paragraphs, each of the components identified as being prone to faults is discussed and include either the solution used to improve fault tolerance or how it might be possible to improve fault tolerance.

A common issue that is present in services(Client and Statistics service) which consume from Kafka is what happens when one of those services goes down? Provided Kafka itself does not fail then this isn't a major issue and data recovery is possible. Kafka doesn't remove messages from its queue so it is possible to use its read offset variable to rewind and retrieve the data which was "lost" while processing. This wasn't implemented into the project as to do so would require additional functionality to store state(which would contain the previous read offset and partition ID which was assigned to the service instance) and then handle the scenario where the partitions get re-balanced amongst the remaining consumers. This functionality would require extensive testing and more importantly time.

If either a Client or Statistics instance fails the current data being processed is "lost" and the client would have to re-submit the query to get the result. This was agreed upon as an accepted failure scenario as the client does not need to pull any data or do any processing. The limiting factor is how fast they can re-submit the query, as it is likely that only a single instance of another service has failed so the new query will go to another service instance.

In the scenario where the Kafka broker itself fails, DSP doesn't necessarily lose data. Kafka attempts to write its queue to disk whenever a new message is published to it. However, if there is a substantial amount of messages being received then it is not guaranteed that those messages will have been written to disk at the time of failure. Disk persistence isn't enough to ensure fault tolerance on the Kafka side.

Here is where using multiple Kafka brokers can help prevent data loss on failure. Multiple Kafka brokers can be used to form a cluster. Then when a consumer connects to a single broker in the cluster it discovers all the other brokers. It's at this stage where the topic replication factor variable comes in. Data can be replicated across brokers in the cluster so that in the event where one of those brokers fails the consumers then switch to consuming from another broker in the cluster. This prevents data from being lost when a single broker fails. This wasn't implemented in DSP due to time constraints. To do so, brokers would be started-up on individual servers and then connected to form a cluster using zookeeper. Consumers then only need to be given the address of a single broker in the cluster to operate.

The Load Balancer service presents a more difficult challenge in terms of Fault Tolerance. In the event of failure, requests would stop passing through the system. The same issue arises with the proposed scaling solution for the Load Balancer service mentioned previously. To work around this a separate service would be required. This service would be very minimal and would be responsible for storing a list of all the available Load Balancer instances and this information would be made available to the Client and Puller services. Each of them would make a request to this service for an available Load Balancer and would return this information in a round-robin fashion to avoid one Load Balancer doing too much work. In the event of a Load Balancer failing the Client or Puller instances would request a new Load Balancer address.

With regard to the Puller's fault tolerance, the most critical component is the database. This project mitigates this risk by using DynamoDB, a fault-tolerant database. DynamoDB is hosted and fully-managed by Amazon. One of DynamoDB's most important features is the ability to replicate databases across availability zones which ensures fault-tolerance. DynamoDB offers database backups and recovery so if a table was lost, it could be restored from a daily backup and any lost data will be restored when the puller polls for new listings.

6 Benchmarks

The statistics processor is capable of processing 200k property messages in under 700ms. This benchmark was averaged across multiple iterations. It focuses on the processing side of the Statistics service and ignore the network constraint of pulling from Kafka.

7 Summary

This project is successful in producing a proof-of-concept of a highly-scalable, fault-tolerant distributed system for computing statistics on large data sets. The system produced allows web uses to make custom Irish property statistic queries efficiently and scalable. The project leveraged REST, Kafka, DynamoDB, and Docker to construct this distributed, scalable, and fault-tolerant system.