Engineering a BigData Pipeline for DotA2

CSCI 5751 - Big Data Engineering and Architecture

NoSQL Proof of Concept

by

Data Diggers

Bhaargav Sriraman - srira048@umn.edu
Prabhjot Singh Rai - rai00027@umn.edu
Suresh Siddharth - siddh010@umn.edu
Vishwesh Mishra - mishr167@umn.edu

Document Date: 1st November 2019

1. Overview

This project's objective is to design and develop a system that can take a look at the data generated by people playing the game of DotA2 and perform analytics-at-scale on it. The aim is to provide general global-level statistics on the game (descriptive analysis), insight on the latest trends followed by the players, as well as look into how the system can provide suggestions and strategy on how to play better (predictive analysis).

1.1 What is DotA2?

A formal introduction to the game from Wikipedia[1] would be a good starting point - DotA2 stands for **Defence of the Ancients 2** and is a multiplayer online battle arena (MOBA) video game developed and published by Valve Corporation. DotA2 is played in matches between two teams of five players, with each team occupying and defending their own separate base on the map (Figure 1.1). Each of the ten players independently controls a powerful character, known as a "hero", who all have unique abilities and differing styles of play.

Figure 1.1 - The miniature map of DotA2

In order to understand the complexity of the game, following details may be helpful. The heroes are the most essential elements of the game. A player needs to pick one out of 117 heroes for the duration of the game. They each have 4 unique abilities that can range from passive effects to "devastating explosions of energy, to complex, terrain changing feats"[2]. Figure 1.2 gives a

representation of the different heroes in the games along with their primary class. In-game equipment or “Items” are another part of the equation. Heroes gain gold as they play the game, and can use these to purchase items. These items provide heroes with bonuses and/or special abilities.

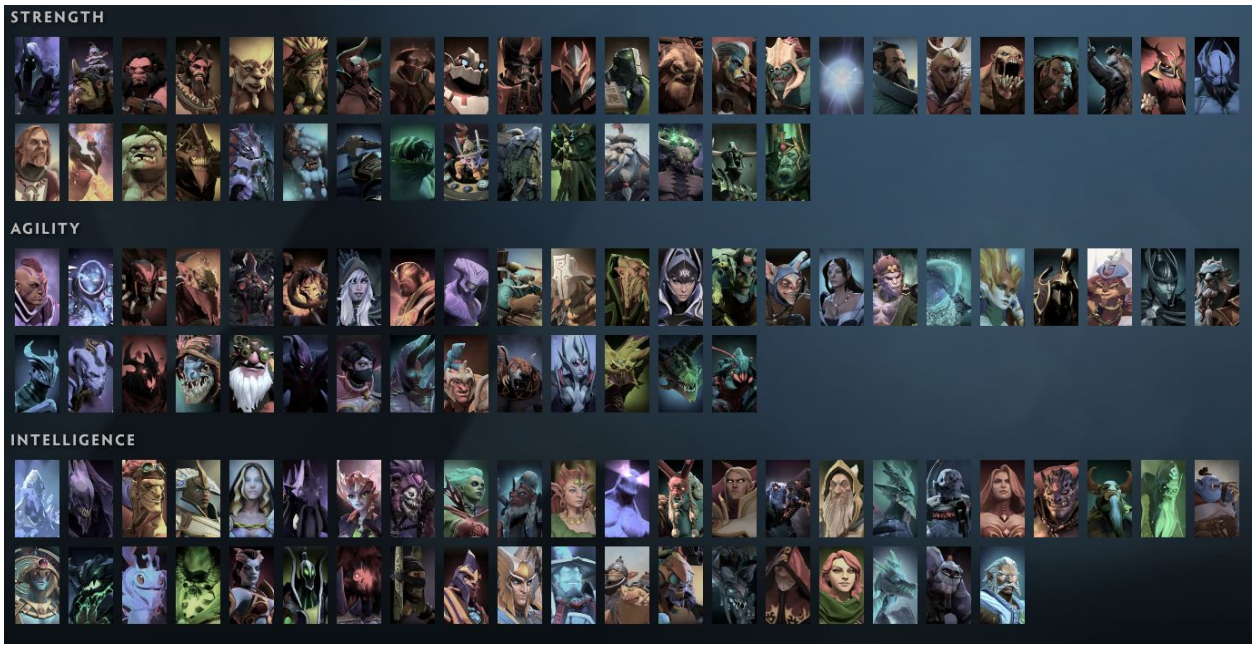


Figure 1.2 - The Heroes of DotA2

The win condition is a lot less complicated to understand. A team wins by being the first to destroy the other team's "Ancient", a large structure located within the enemy base.

Note: More information on Dota2 in Appendix I

1.2 Business Questions

After having an understanding of a few of the core concepts of the game, here are the business questions that have been answered:

Easy business questions

1. What is the longest game played?
2. What is the average duration of a game?
3. What was the average first blood time?
4. Has a match been abandoned?
5. How many games were played in a day?

Medium business questions

1. How many games has a particular hero been played in the last week?

2. What is the win rate of a hero in the last week?
3. What is the overall win-rate of a hero?
4. What are the most effective items for a hero?

Hard business questions

1. Identify pairs of heroes that work well together. Quantify the synergy by win rate.
2. Predict the outcome of a game, given the heroes playing.

Ambitious business questions

1. Include a replay parser in the data pipeline. What new data do we get? Can we enhance any of the previous business questions? *(This question has not been answered within the given time-frame of project submission)*

1.3 Why is this a Big Data problem?

DotA2 has been a game that has been around since 2011 and is still going strong. With a global player base, it has people playing the game around the clock. As of last month, the game has 9.9 million unique users login to the game[1] and averages around 400k people playing the game at any point in time. This gives the first V of Big Data - *Volume*.

It is also the nature of the game that there are times at which split-second decisions are made and hence the solution proposed would also need to respond just as fast in order to satisfy the ultimate goal of providing real-time tracking and support, giving the second V of Big Data - *Velocity*.

Currently, data about the game is ingested using the identified data source which would be discussed in coming sections. However, if possible, actual game replays can be downloaded and parsed to generate a new dataset. This would introduce a new format of data and brings in the third V of Big Data - *Variety*.

Different data sources were identified which could provide the data about the game. Some were processed by the third-party but one of them had data in its native format and was provided by the authentic data provider i.e. by game developers themselves. The choice of this data source introduced the fourth V of Big Data - *Veracity*.

[1] As per in-game dashboard as of October 3rd, 2019

2. Data Source

When exploring the data sources, two in particular were found which could answer the business questions.

	OpenDota[3]	**Steam[15]**
Definition	OpenDota is an open-source platform that provides an API to fetch DotA2 related data.	SteamAPI is hosted by Valve Corporation, who are the creators of DotA2. They provide information on completed DotA2 games.
Number of Attributes	Data provided by OpenDota provides a higher number of attributes (106) in a single call	Native Steam API provides less number of attributes (30).
Originality of Data	OpenDota employs a replay parsing mechanism to obtain a richer data set. Attributes generated are added to the original data.	Steam's API is the original source of data and provides raw match data only
API Constraints	50,000 requests for OpenDota API in free tier per month	100,000 API calls per day for Steam API

Other notable sources which were found but were ranked lower because of the following issues:

i. Kaggle - Not considered as it was a very old dataset.
ii. UCI ML Library - Same issues as with Kaggle.

2.1 Choice of API

After going through both the APIs, Steam API seemed more suitable because of the following reasons:

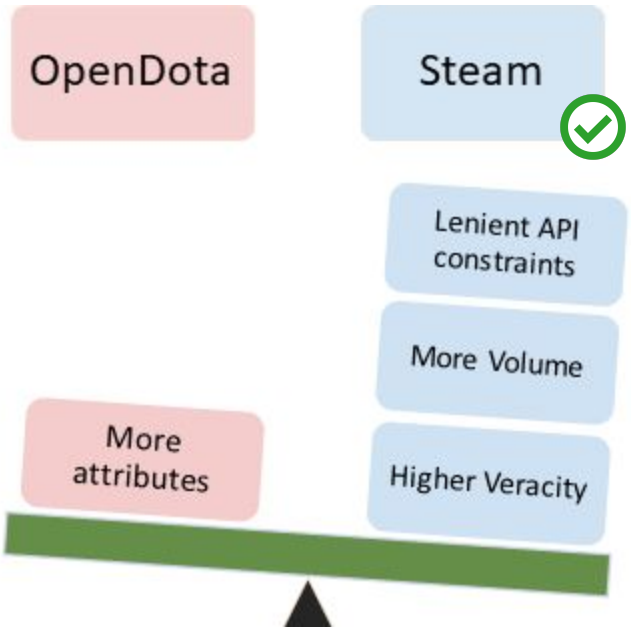


Figure 2.1 - "Steam" Overweighs "OpenDota"

- Steam API's veracity is higher than the OpenDota dataset as the former is provided directly by the game's developers.
- More data can be fetched from Steam's API than OpenDota. Although OpenDota may give greater number of attributes, having more data is crucial as the volume is more critical for solving the business problems than the number of attributes.
- An open-source DotA2 replay parser (Manta) is also an option to complement the data pipeline.

2.2 Constraints of using Steam's API

1. Obtaining a Steam WebAPI key requires an account that has at least one paid game. Initially we had one such account but as multiple calls are required at the same time, more API keys were acquired.
2. We will need to manually limit our requests to one request per second per key to reduce the strain on the servers.
3. In case we get a 503 error, the matchmaking server is busy or we exceeded limits. This kind of errors will have to be handled in code[4]
4. Finally, we are limited to 100,000 API calls per day under the Steam API conditions[5].

2.3 About Data

Data from the Steam API needs to be fetched in two stages:

1. **Fetch the Match IDs**

   The first stage fetches the Match IDs which have been completed recently. This data is fetched from the end-point:
   `http://api.steampowered.com/IDOTA2Match_570/GetMatchHistory/v1?key={{API_KEY}}`

   A sample response of this API can be found here. The important field we are looking for here is the list of `match_id` fetched from the `matches` list. This will form the basis of making the next step of data acquisition.

2. **Fetch the Match Details for each ID**

   The second stage fetches full match details given a particular `match_id`. The data is pulled by making a call to the end-point:
   `http://api.steampowered.com/IDOTA2Match_570/GetMatchDetails/v1?key={{API_KEY}}&match_id={{match_id}}`

   A sample response of this API can be found here. This data forms the major portion of the data pipeline and most of the transformations and denormalization described in the next steps are based on this data.

2.3.1 Important Fields

1. Match Duration: This field denotes in seconds the number of times elapsed before a game came to a conclusion. The conclusion could either be by a team winning, or an entire team abandoning a game
2. First Blood Duration: This field denotes the time in seconds during which a hero was first killed. This is a good indicator of which team might have the advantage in the early stages of the game
3. Hero ID: Hero ID is a attribute of a player and denotes which one of the 117 heros was played by that person
4. Leaver Status: Leaver status is another attribute of a player and lets us know if a player abandoned a match or not. More details on the values of this field can be found here under the leaver status.
5. Game Mode: Game modes are a set of restrictions within which the game of DotA 2 can be played. More can be found in the appendix.
6. Radiant win: This field denotes which team won this particular game, `true` being Team Radiant won the game and `false` being Team Dire won the game,

2.4 Inconsistencies in data

The data for the most part remained consistent. However, there were situations where some errors were faced like "*Too many requests, please try again in a while*", as well as blank responses from the server which were handled in code.

Another issue faced was the change in Steam API data through the duration of the project. However, since there was no data removed and only new fields added, it did not impact the execution of the project. That being said, new insights which could be drawn from the data were missed out in this project.

3. NoSQL Storage Tech

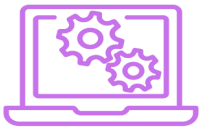
FaunaDB was chosen for the purpose of processing, storing and analyzing DotA2 data. FaunaDB is an indexed document store that allows access to the stored data using multiple models - relational, document, graph and key-value.[6] FaunaDB core functions are inspired by Calvin: a clock-less, strictly-serializable transactional protocol for multi-region environments.[7]

3.1 Features[8]

Multi-API querying

- FaunaDB offers polyglot APIs for data access, including native GraphQL and FQL for advanced tasks and more productive development.
- ***Feature usage:*** FQL features like join, map, lambda and batch querying were exploited in the project instead of implementing it at the application level.

Powerful indexing



- A unique approach to indexing makes it simpler to write efficient queries that scale with your application.
- ***Feature usage:*** FaunaDBs approach to indexing has good benefits. The indexing methods are described in detail in Appendix III.

Built-in temporality

- Travel back in time with temporal querying. Run queries at a point-in-time or as change feeds. Track how the data has evolved.
- ***Feature usage:*** This can be used for live-game tracking. It has helped answer business questions related to the start and end of an event in-game, as well as dealing with range queries over time.

---

4. Database Infrastructure and Setup

After analysis it was inferred that FaunaDB can be implemented in 3 different ways:

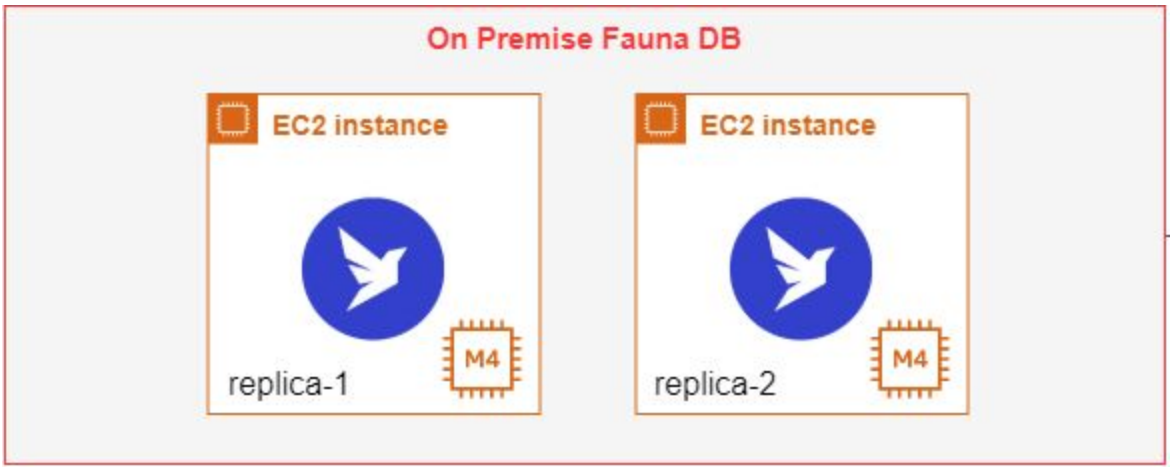
1. **Use the public Docker[2] image**
   The significance of the docker approach is simplification and acceleration of deployment and configuration of FaunaDB. However, some of the key features which are available in the cloud platform are not supported in the Docker. For example, GraphQL is not supported in the dockerized version.

2. **Use the Cloud PaaS**
   FaunaDB also provides a cloud platform to build an application. Although all the features are present in it, it is not free. This is a problem if the application needs to be considerably scaled in the future.

3. **Use Fauna Hybrid**
   Fauna Hybrid involves installing the database directly on the host machine using packages provided by Fauna. This would involve infrastructure maintenance becoming our responsibility, however, this is the most stable format for a POC.

4.1 Choice of Infrastructure

The final decision on infrastructure was to go with the Fauna Hybrid approach. Through trial and error, this was the most stable method we found short of spending money for the managed cloud system.



[2] Docker is a tool designed using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one unit.

The database was implemented as a single cluster, with 2 replicas on an AWS cloud environment. Each replicas was deployed on a separate EC2 instance with 16GB of RAM. However, this configuration was done primarily to minimize the cost along with achieving acceptable stability.

In an ideal production environment, we would prefer to go with a multi-cluster approach with a minimum of 3 replicas per cluster in order to get the best performance, and stability. More information on suggested hardware requirements and region split-up can be found in the official documentation.

4.2 Documentation Notes

1. Recommended requirements for FaunaDB
2. Setting up a cluster in AWS

4.3 Challenges Faced

Through the duration of the project, many issues were faced when working with FaunaDB and came up with various fixes, configuration changes, and code changes.

1. **Working with FaunaDB on Docker**
   The first approach to setting up FaunaDB was to set it up using multiple Docker containers connected to each other. While this was easy to set up, it did not meet the mark with system stability. Containers went offline frequently and at times even took data along with it.

   In one such instance, even though we were able to resolve issues and bring the containers back up, they were not able to communicate with each other resulting in a permanently split cluster which was no longer usable. In this instance, data was recovered through our HiFi data backup using the following code.

2. **FaunaDB performance**
   The documentation for minimum requirements specifies systems with 16GB RAM. This was meant in the true sense. The cluster which was setup was barely able to handle the load that the data pipeline put it through. However, with provenance tracking and monitoring, and application retry logic, most of these issues were mitigated.

   That being said, these issues should not occur if a cluster is setup keeping the recommended infrastructure requirements in mind.

3. **Working with FQL**
   FQL stands for Fauna query language and is the native language through which queries into the database are supported. Though it had extremely powerful concepts to the level of supporting joins in the database, it is extremely non-intuitive. A large amount of time

was spent trying to understand how to do simple operations (from an SQL stand-point) via FQL.

Some functionalities of FQL are language specific. For example, ‘count’ and ‘range’ querying is not supported in python. This proved to be a hindrance as a switch from python to JavaScript was required.

If a decision is made to move forward with using FQL, it is vital to allocate time to getting an in-depth understanding of how it works in order to prevent the introduction of inefficient queries. This was a problem that was widely prevalent over the initial duration of the project.

Note: While Fauna recently started supporting GraphQL, this is not supported in the Docker version where the project ran on most of the time until the switch to Fauna Hybrid, at which point, switching to GraphQL was not feasible. This is the reason the project report does not take into consideration the advantages/disadvantages of GraphQL.

Note: Fauna Community on slack proved to be highly supportive during the project.

Further information on FaunaDB Clustering concepts can be found in Appendix IV

5. Project Life-cycle and Architecture

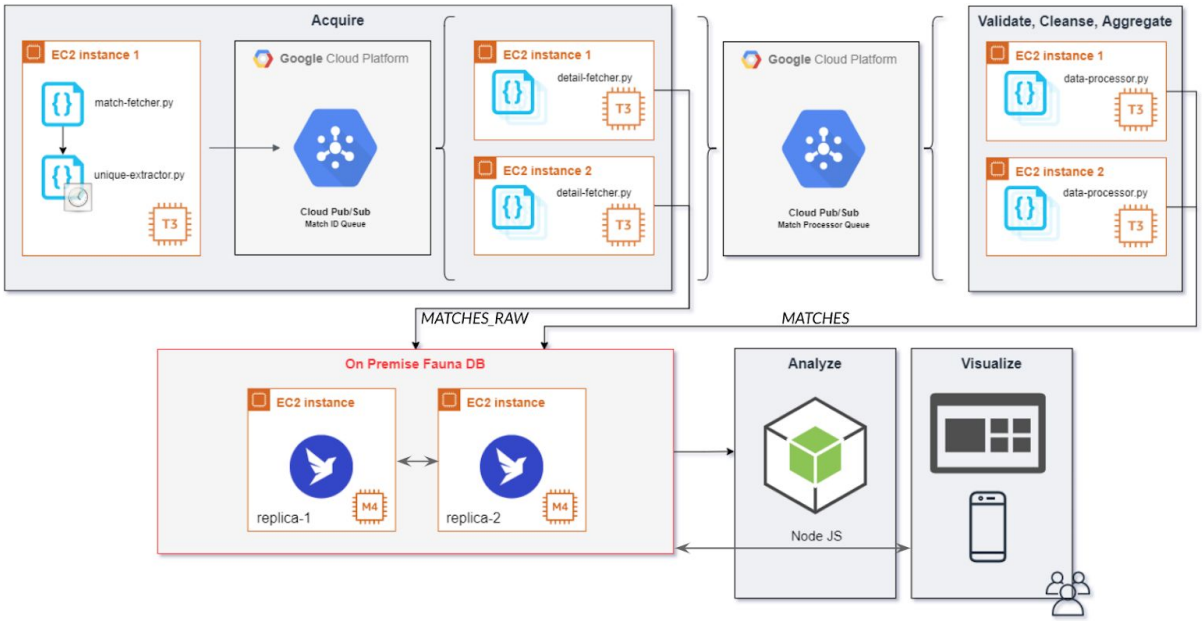


Figure 5.1 - Project Architecture

The architecture followed the following crucial stages of the Big Data Life Cycle:

5.1 Data Acquisition and Filtering

Stage 1 - Fetching Match IDs for games

As described in the previous portion of the report, data acquisition needed to be done as a two stage process.

Process

The first stage was to fetch the Match IDs of recently completed games by querying data from Steam's API endpoints. Match IDs were extracted from the response and stored into a file. There is a limit of one call per second on the API used, and a delay had to be introduced at this stage to prevent the API from locking out the API key being used.

The next step in the process was to extract unique Match IDs from this list and pass it on to the next stage of the data pipeline. The data hand-off was done by using a message bugger - Google PubSub.

Learnings and Challenges

The major challenge faced in this stage of the pipeline was the handling of duplicates. The API provides a feature to impose an order on the Match IDs being fetched. However, it did not work as expected which resulted in accumulation of duplicate Match IDs during each of the subsequent calls of the API. The initial attempt at this led to a very large duplication factor (5 billion Match IDs fetched consisted of only 250k unique values). In order to work around this, a two step process was introduced to fetch data.

1. A separate API call[3] was utilized to get historical data into the system in order to provide an initial seed of data on which the rest of the pipeline could be built. This API supported ordering and was instrumental in getting the project rolling. However, this API could not be used to fetch live information.

2. Handling the duplication issue in live data necessitated the second stage of this process which introduced the cron functionality, where data was filtered for unique values every minute after fetching data and then passed on the rest of the pipeline. While this still had the potential to introduce duplicates, it brought down the percentage of duplication to a level where the load of handling duplicates could then be passed on the database layer.

   This approach was favoured over completely letting the database handle it in order to reduce the overall load on the database, as well as to make CPU utilization more efficient. The initial filter could be processed while the thread waited for the next window to make the API call.

3. Introducing a Message Buffer was also not part of the initial project design and was introduced later in order to come up with a design which could make the system scale better as well as decouple the initial data fetching process to make it as resilient as possible. Before deciding on a message buffer, a custom built queue solution was tried and discarded in favour of the more durable cloud solution. The custom solution was not as durable (can only run on a single system) and could not be scaled (file-based data handoff) to the levels a message broker could.

Stage 2 - Fetching Match Details for a Match ID

The next step of the acquisition process was to fetch the Match IDs published by the previous step and to get the full details of the instance of the game played.

Process

This was implemented by having a fleet of 12 consumers which would pull Match IDs from the PubSub topic, hit the Steam API to fetch match details for the Match ID and persist the raw data

[3] `https://api.steampowered.com/IDOTA2Match_570/GetMatchHistoryBySequenceNum/V001?key={{API_KEY}}`

into FaunaDB (Collection: `matches_raw`). Once data was persisted, the Match ID was pushed to a separate queue to start the data processing stage for that record.

This stage also introduced the first provenance record into the data, which tracks the start time, end time, and the overall API call time duration to the Steam API. This was a vital metric to be monitored to prevent any external system failures resulting in the failure of the pipeline.

This approach was followed as the data hand-off only needed to be the Match ID instead of the entire Match Details JSON which would have been a large document.

Learnings and Challenges

The major learning of this stage was mainly on the number of instances which could be deployed. A balance had to be maintained between the number of messages produced by the previous stage and the processing time of this stage.

Fault tolerance was introduced here by having the any timeout or blank response exceptions to publish the Match ID back to the processing queue.

5.2 Data Validation, Cleansing and Aggregation

Stage 3 - Data Transformation

This stage deals with the different filters and denormalization which were done on the raw data in order to optimize the solutions for our business questions.

Process

This stage was implemented by having a fleet of 6 consumers fetching Match IDs which have been persisted in the database and running transformations on it. The steps in this stage can be broken down into Data Filtering and Data Processing substages.

Data filtering was done on the basis of prior knowledge about the game. The filters applied on the data were:

1. Ignore any practice matches which come into the pipeline. (code)
2. Ignore any matches which have duration as 0. (code)
3. Ignore any matches which do not have a full set of 10 heros. (code)
4. Ignore any matches played on 'fun' modes. (code)

The data processing substage mainly dealt with optimizing data to answer our business questions.

1. Update an aggregate collection (`match_aggregate_info`) to hold meta-data such as total count of records, maximum; minimum; and average match duration, average first blood duration, maximum first blood duration. (code)

2. Create a new document in the `matches` collection which holds basic information about a match and also track if the match has been abandoned by iterating through the player status of each player in the game. (code)

3. Process data about the heroes played in the match, and update the information on the number of matches the hero features in, the number of wins, and update statistics on item preferences. This was done on the `heroes` collection. Item preferences were tracking by having an array in the Document for the number of times an item was purchased by the hero, and updating the same every time we processed a match with the items in the heroes inventory. (code)

4. With the previous step taking care of overall information a separate collection was created (`heroes_temporal`) to keep track of the win/lose statistics which could be queried as a function of time. This collection had a record for every time a hero appeared in the game and kept track of the Hero ID, Win/Lose, and the Start Time of the match. (code)

5. Create a feature vector for the match which contains a vector having all the heroes which featured in the game and the `radiant_win` flag. This information is stored in the `match_prediction` collection and will be queried once a day to create a ML Model which can run predictions. (code)

6. Process information on hero pairs. This data is stored in an aggregate table with a special key design to track all possible hero pair combinations and update win/lose statistics for the hero pair. (code)

7. Introduce further provenance tracking in order to measure end-to-end processing time of records. (code)

Further information on how the business questions were answered in given in the next part of this document.

Learnings and Challenges

The first challenge of this stage was due to the incremental approach to building the data pipeline. By the time the data processing code was ready, the pipeline had already ingested close to 2 million data points with more in the queue. This data had to now be put through this stage. This was handled by writing a stand-alone program to read existing information in the database and publish their Match IDs to the queue so they could be processed.

The second, more critical challenge faced was the approach to query design initially used. The first iteration of the data processor took almost 3 seconds to process a single match and was not feasible in the data pipeline. The major issue with the approach followed was to run transaction-like queries to create and update records which took a lot of time.

A major redesign was done to process data inserts and updates in bulk in order to speed up the entire process. A sample of this can be seen in the following code snippet where all the meta-data aggregations are pulled in a single query. The full for this process can be found here.

```
aggregate_info_list = client.query(
  q.map_(
    q.lambda_(
      'data',
      q.get(q.ref(q.collection('match_aggregate_info'), q.var('data')))
    ),
    [
      getIntValue('min_match_duration'),
      getIntValue('max_match_duration'),
      getIntValue('max_first_blood_time'),
      getIntValue('mean_match_duration'),
      getIntValue('avg_first_blood_time')
    ]
  )
)
```

With the help of these transformations, the overall processing time was reduced from 3 seconds to ~300ms per record.

Note: This learning was a result of both this and the analysis stage, but since the changes belong here, it has been included in this section.

5.3 Data Analysis and Visualization

Stage 4 - Data Analysis

This stage consists of the queries which were run on top of the transformations done on the previous stage to get the answers to the business questions.

Process (code)

This stage was implemented using the NodeJS library of FaunaDB. The first step of the process was to create the indexes required to answer the business questions. In-Depth information on the index created and their rationale can be found in Appendix III. Due to the nature of the data modeling done in the previous step, most of the business questions could be done via the help of simple queries to the database, however, certain business questions required some application level logic to answer.

Learnings and Challenges

The first version of this stage was attempted using the Python code library for FaunaDB. However, in time it was found that the Python library does not support the complete functionality of Fauna. Certain key functions like Count and Range were not supported in the Python library, but were implemented for Javascript. Due to this fact, the language for this stage of the pipeline is not Python which was the standard up to now.

Stage 5 - Data Visualization

For visualizing the solutions built so far, along with real-time information such as the number of data items being processed per second and added to the database, a data visualisation layer was created to visualize information in real-time.

Process

A backend service was created in Node and deployed in an AWS instance. This service queries data in real-time and publishes information to Pusher, which sits as a real-time layer between the server and clients.

A simple user-interface was created using ReactJS and deployed to 'dota.prabhjotrai.com' to visualize the business questions.

Note: Currently the UI only supports some of the easy business questions. Including all the business questions as a complete experience would require a lot of time in UI design and implementation which was not possible within the duration of this project. However, all business questions solutions are built to be compatible with this dashboard.

Learnings and Challenges

The main challenges of this phase were:

1. One important detail considered in the design of the stage was limit the number of connections to the database, as well as the network traffic involved in pushing the results of queries to the end user. Another consideration was to improve locality. Rather than having end users from potentially all over the world query our database, this approach can utilize a server located close to the database to query and fetch the final information. This information is a lot smaller in size, and can be transferred to the end user using edge network which are optimized for this task.
2. The next learning was on domain knowledge. The team was not familiar with creating a NodeJS application and upskilled to implement the dashboard. Learning Javascript ES6 was another tedious task.

Note: Further information on the setup steps and requirements of the application code can be found in Appendix II.

6. Data Models and the Business Questions

This section gives more detail on the different collections and how they were used to answer the business questions.

The collections used to answer our business questions are:

6.1 Match Aggregate Info

The purpose of this collection is to store the aggregate values while processing the data in order to avoid a full table scan.

6.1.1 Business questions answered?

1. What is the longest game played?
2. What is the average duration of a game?
3. What was the average first blood time?
4. How many games have been played?

6.1.2 How are the Business Questions answered?

Each of the records in the collection stores the answer to a question. A single read operation is used to fetch the data.

6.1.3 Design information

Since FaunaDB only accepts Integers as keys, there was a need to convert meaningful strings to integer values. This was done by adding the ASCII values if the character in the strings using the following code snippet:

```
def getIntValue(key):
   sum = 0
   for char in key:
       sum += ord(char)
   return sum
```

The collection has 6 records:

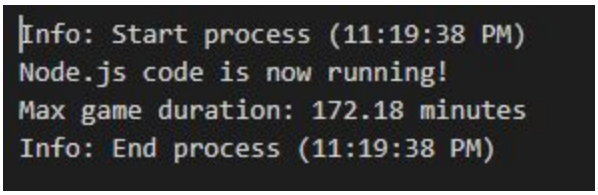
1. `min_match_duration` (key = 1909) - **Stores a floating point value**
2. `max_match_duration` (key = 1911) - **Stores a floating point value**
3. `mean_match_duration` (key = 2002) - **Stores a floating point value**
4. `max_first_blood_time` (key = 2122) - **Stores a Integer**
5. `avg_first_blood_time` (key = 2114) - **Stores an Integer**
6. `match_count` (key = 1173) - **Stores an Integer**

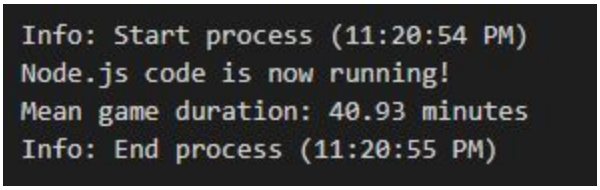
The data processor checks if these records need to be updated after processing every match info. The count is incremented every time.

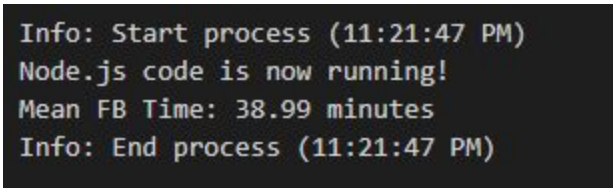
6.1.4 Data Sample

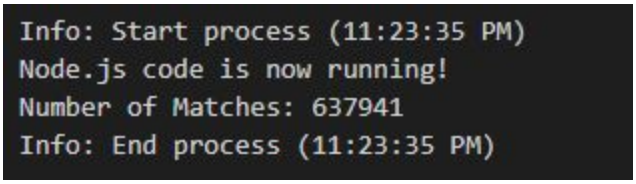
```
{
  "ref": q.Ref(q.Class("match_aggregate_info"), "2002"),
  "ts": 1572657903105000,
  "data": {
    "data": 2456.3720422700185
  }
}
```

6.1.5 Results









6.2 Matches

The purpose of this collection is to store validated and processed match information.

6.2.1 Business questions answered

1. Has a match been abandoned?
2. How many games were played in a day?

6.2.2 How are the Business Questions answered?

The data contains a flag, `abandoned_status`, which is used to identify abandoned matches.
An index, `match_by_ts`, is built to search through the collection to obtain games played in a day.

6.2.3 Design information

This collection uses the Match ID as the key since the business questions answered are specific to a particular match only. Another important part of this collection that is has all provenance information regarding the processing of the record.

Each entry in the collections has the following fields:

1. `abondoned_status`: To identify an abandoned match - **Boolean**
2. `radiant_win`: Identifies the winning team - **Boolean**
3. `start_time`: Start time of the match - **Timestamp**
4. `match_id`: ID of the match - **Integer**
5. `number_of_bans`: Number of heroes banned from being picked- **Integer**

6.2.4 Data Sample

```
{
  "abandoned_status": false,
  "radiant_win": true,
  "start_time": q.Time("2019-07-28T07:02:28Z"),
  "match_id": 4930883420,
  "number_of_bans": 1,
  "provenance": {
    "dataFetchStage": {
      "startTime": "2019-10-29 14:56:18.469366",
      "apiCallDuration": "0:00:00.226710",
      "processedBy": "fetcher-7"
    },
    "dataProcessStage": {
      "startTime": q.Time("2019-10-31T00:10:59.236955Z"),
      "processDuration": 857520,
      "processName": "data-processor-1"
    }
  }
}
```

6.2.5 Results

```
Info: Start process (11:24:27 PM)
Node.js code is now running!
Is match abandoned (4929685690): false
Info: End process (11:24:28 PM)
```

```
Info: Start process (11:25:13 PM)
Node.js code is now running!
Number of matches in the last 4 months: 468713
Info: End process (11:25:32 PM)
```

6.3 Heroes

The heroes collection has aggregate information specific to each particular hero. It holds information such as number of matches appeared in, number of matches won, and item preferences.

6.3.1 Business question answered

1. Overall win-rate of a hero
2. Top items bought by a hero

6.3.2 How are the Business Questions answered?

In order to answer the business questions it is not feasible to scan through all the records of the `matches` collection. As a result, data particular to a hero was denormalized and stored separately in a heroes collection. This collection is indexed with `hero_id` which ranges from 1 to 129.

6.3.3 Design Information

Each entry in the collections has 5 fields :

1. `name` - Name of the hero - **String**
2. `id` - Hero ID - **Integer**
3. `games` - Total number of games featured by the hero - **Integer**
4. `wins` - Total number of games where the hero featured in the winning team - **Integer**
5. `items` - An array whose index corresponds to the `item_id` and the value corresponds to the number of times the item was bought by the hero - **Array of Integers**

As a match is being processed by the data processor the following details are identified:

1. Heroes in the winning team
2. Heroes in the losing team
3. Items bought by each hero

Once identified, the counts are then updated to corresponding fields of each of the 10 heroes involved. This process is made faster by using batch read and batch update queries supported by FaunaDB. The DB is hit only twice, once to read 10 records and once again to update 10 records.

To answer the business question,

1. The win-rate of a hero is obtained by fetching the record of the hero and dividing wins by games.
2. The popular items bought are extracted by sorting the items array along with the indexes.

6.3.4 Data Sample

Since the JSON is long, it is not included in the report and can be found here.

6.3.5 Results

```
Info: Start process (11:30:14 PM)
Node.js code is now running!
Overall win-rate of the Hero (Viper):  53.010000000000005
Info: End process (11:30:15 PM)
```

```
Info: Start process (11:33:59 PM)
Node.js code is now running!
Viper - Top Item 0: rod of atos
Viper - Top Item 1: wraith band
Info: End process (11:34:00 PM)
```

6.4 Heroes Temporal

This collection was created in order to answer questions about hero information within the time domain

6.4.1 Business questions answered

1. How many games has a particular hero been played in the last week?
2. What is the win rate of a hero in the last week?

And index was created to help search through the collection efficiently. The index used here is hero_temporal_winrate.

6.4.2 How are the Business Questions answered?

To answer the business question,

1. Given a Hero ID, find the total number of games played is calculated by performing union over both, wins and loses. (self join query)
2. For the same Hero ID, calculate the number of wins by query with `id` and `win`
3. Calculate the win rate in application code by dividing the two values.

6.4.3 Design Information

A record is inserted for every hero when processing a match along with the timestamp.

Each record in the collection has the following fields:

1. `id` - Hero ID of the hero associated - **Integer**
2. `win` - Whether the hero won that game or not - **Boolean**
3. `match_start_time` - start time of the match - **Timestamp**

6.4.4 Data Sample

```
{
  "ref": q.Ref(q.Class("heroes_temporal"), "247529952152388096"),
  "ts": 1572321807910000,
  "data": {
    "id": 47,
    "win": true,
    "match_start_time": q.Time("2019-07-27T16:14:49Z")
  }
}
```

6.4.5 Results

```
Info: Start process (11:27:36 PM)
Node.js code is now running!
Number of games played by Hero (Viper) in the last 5 months:  63111
Info: End process (11:27:45 PM)
```

```
Info: Start process (11:29:10 PM)
Node.js code is now running!
Win Rate of Hero(Viper) in the last 5 months:  52.96999999999999
Info: End process (11:29:22 PM)
```

6.5 Hero pairs

This collection stores aggregate information on statistics of how pairs of heroes perform. During the start of the game, heroes usually ‘lane’ together as pairs and try to defeat the opposite teams heroes. This makes this information important as having a pair with good synergy would increase the chances of a favourable outcome.

6.5.1 Business questions answered

1. Identify pairs of heroes that work well together. Quantify the synergy by win rate.

To find pairs that work well together, it was decided to use win-rate to measure the synergy between them. This means that a hero works best with the hero it has the highest win ratio with.

Note: This was first implemented in the heroes collection. In this implementation, for each hero, an array of winning heroes and losing heroes needs to be updated. However, there was a problem with this approach. Every time a hero is processed, around 30 values were needed to be updated. Apart from that when multiple threads are running simultaneously, it is highly likely that two or more of them are updating the record of the same hero. This will lead to inconsistency if we read the data in a soft state. As a result, it was not preferred to increment and store the values.

6.5.2 How are the Business Questions answered?

1. All the possible key combinations which involve the particular hero id is generated.
2. Fetch all the records in a single batch read
3. Sort the result based on the win-ratio
4. Return top results as request by the user

6.5.3 Design Information

Each entry in the collection has the following fields :

1. `wins`: Total number of games won where the pair were in the winning team - **Integer**
2. `games`: Total number of games where the pair were in the same team. - **Integer**

As a match is being processed by the data processor the following details are identified:

1. Heroes in the winning team
2. Heroes in the losing team

All possible pairs of heroes are then updated with new values. In this model, the chances of two threads updating the same pair of heroes are smaller. Compared to the earlier suggested process, it has only two writes per hero pair.

To identify the records uniquely, a key design is introduced.

1. Since there are 129 heroes in total, the hero ids are first converted to a 3 digit string.
2. The smaller hero id is then concatenated with the larger hero id to obtain the application level key for the collection.
3. Fauna internally converts this into an integer, be removing any leading zeroes and uses it as the actual key. (‘001003’ will be converted to 1003)

6.5.4 Data Sample

```
{
  "ref": q.Ref(q.Class("hero_pairs"), "1007"),
  "ts": 1572660892610000,
  "data": {
    "games": 7770,
    "wins": 3884
  }
}
```

6.5.5 Results

```
Info: Start process (11:36:33 PM)
Node.js code is now running!
Viper - Top Hero Pair 0: spectre with Win-Rate: 0.5962
Viper - Top Hero Pair 1: skeleton king with Win-Rate: 0.5738
Viper - Top Hero Pair 2: mirana with Win-Rate: 0.5616
Viper - Top Hero Pair 3: windrunner with Win-Rate: 0.5565
Info: End process (11:36:34 PM)
```

6.6 Match Prediction

This collection stores the feature vectors of a match using the start_time of the match. The feature vector consists of 259 columns. The first 129 columns are the representation of hero_ids for the first team, the next 129 columns are the representation of hero_ids for the second team. The last column is the indicator if the radiant team has won or lost.

6.6.1 Business questions answered

1. Predict the outcome of a game, given the heroes playing

For a given match, the information of heroes playing on both the teams are known. Given different matches with different combinations of heroes in each of them, the outcome of any new match is to be predicted in terms of win or loss of the Radiant team.

6.6.2 How are the Business Questions answered?

Feature Vector Details

1. A feature vector consists of only 0s and 1s in each column.
2. Although there are only 117 heroes, the point of taking 129 columns is due to the fact that the hero ids are not consecutive i.e. last hero id is 129.
3. The first 129 columns of a feature vector have only 5 1s depending on the heroes chosen by the radiant team. The next 129 columns also have only 5 1s depending upon the heroes chosen by the dire team.
4. The last column is 1 if the radiant team won the match and 0 if it has lost it.

Steps involved in answering the business question: (Code)

1. Some feature vectors of different matches are fetched from the match_prediction collection using `all_match_prediction` index. (Currently matches are fetched based on the number of pages specified during the batch process. This could be modified to querying matches based on a time window (indices have been created for this) but it would then require range querying in javascript and training models in python and hence back and forth switch from python to javascript. Although the training process can

always be enhanced and pruned seamlessly, the current model training process can tackle the business question at hand)

2. A feature matrix and win-labels (radiant win/loss) are extracted from these feature vectors.
3. A machine learning model is trained using a Support Vector Machine classifier with this feature matrix and win-labels.
4. The trained model is saved in data directory as a python pickle file.
5. This model could be used for prediction of the outcome of a match using the predict functionality of the classifier stored.

6.6.3 Design Information

Each record in the collection has following fields:

1. `start_time`: This represents the start time of the match for which the feature vector has been stored - **Timestamp**
2. `vector`: This represents the feature vector of a match - **List of Integers**

6.6.4 Data Sample

```
{
  "ref": q.Ref(q.Class("match_prediction"), "247644261519983104"),
  "ts": 1572430821820000,
  "data": {
    "start_time": q.Time("2019-07-28T07:56:48Z"),
    "vector": [0, 0, 1, 0, 0, 0, 0, 0, 1,...]
   }
}
```

6.6.5 Results

```
/home/vishwesh/anaconda3/bin/python /home/vishwesh/BigData/dota-pipeline/data_dump/test_pickle.py
Trained Model parameters:
Kernel:  rbf
Win Labels: [0 1]
Gamma: auto
Model Accuracy =  84.0 %
```

With this we have answered all the business questions proposed in the document.

7. Concluding Remarks

This project involved a lot of learning, hard work, failures, and eventually success. The major learnings are listed below:

1. Docker is a good tool for initial setup for a technology and get the hands dirty, but it may not be the most suitable for production environments
2. Pushing real time data to frontend applications is more efficient when there are a lot of users using the frontend
3. While designing any application, we need to keep in mind the scalability of our architecture. Would it be possible to add more instances of the application without any effort? Would it help?
4. Learnt the importance of provenance and how useful it can be to store it (allowed us to recover from multiple failures)
5. Decouple different stages of the architecture so that each stage can be approached as a separate module.

Appendix I

About DotA2

DotA2 is a massively multiplayer online based battle arena game that is considered to be one of the biggest games in the esports (electronic sports) industry. There is a lot of money being put into the ecosystem and DotA2, in particular, has the distinction of having the tournament with the highest prize pool of all time of $34.3 million. The below figure shows estimates from Newzoo's 2019 esports market report.

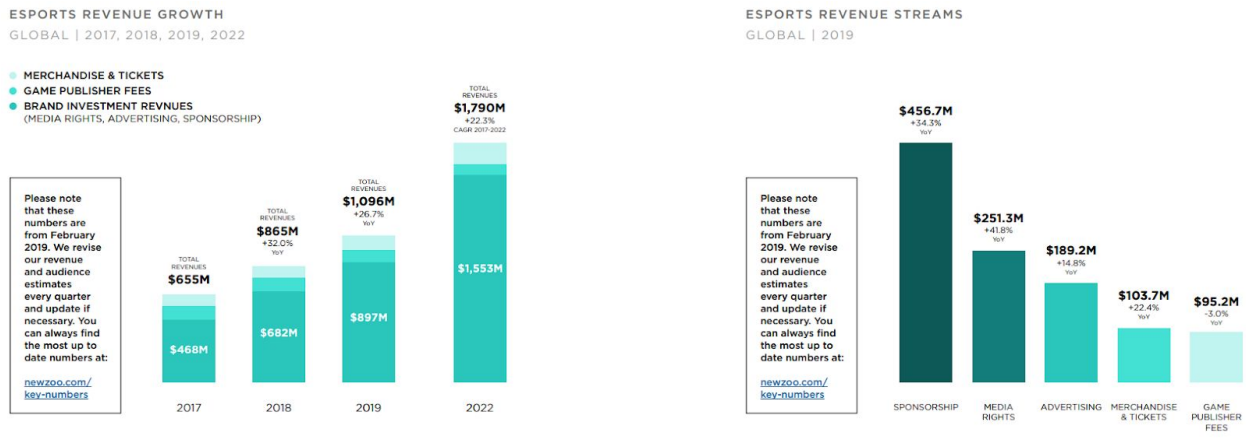


Figure - Newzoo's 2019 Esports Market Report

Heroes[1]

Heroes are divided into two primary roles, known as the core and support. Cores, which are also called "carries", begin each match as weak and vulnerable, but are able to become more powerful later in the game, thus becoming able to "carry" their team to victory. Supports generally lack abilities that deal heavy damage, instead having ones with more functionality and utility that provide assistance for their carries, such as providing healing and other buffs. Players select their hero during a pre-game drafting phase, where they can also discuss potential strategies and hero matchups with their teammates.

Items

There are close to 200 items in the game that are built on the 10 heroes that are in the game at different times. Items are crucial to how the game is played and at times can be as important and powerful as a hero's ability.

Ranked Matchmaking

Matchmaking is the process through which the system groups players into opposing teams for public games. With the exception of Bot games, matchmaking is mostly determined by matchmaking ratings (MMR).[12] There are two kinds of match making, normal and ranked. Only All Pick and Captains Mode are available for ranked matches.

Game Modes

Game modes are a set of restrictions within which the game of DotA 2 can be played. Players are given the option of picking what game modes they want to play. Current game modes for public matchmaking include:

- All Pick
- Captains Mode
- Single Draft
- Random Draft
- All Random
- Ability Draft
- All Random Deathmatch

Captain's Mode[13]

Captains mode is the standard format for tournament games. The captains ban certain heroes, up to six per team, preventing either team from picking the hero. The captain also chooses five heroes for their team. After the captains choose five heroes, each player chooses a hero from their captain's selections. The starting team is randomly selected in matchmaking; if playing in a private lobby a starting team may be specified. Some heroes that were recently created or tweaked are unavailable in Captains Mode; they will be added eventually.

All Pick[14]

In the All Pick mode, players can choose from the full hero pool. Players can choose any hero, random a hero, re-pick a hero, or swap heroes with teammates. Teams alternate picking. Whenever it is a team's turn to pick, anyone on that team can pick their hero. Once a selection is made, it immediately switches to the other team to pick. Initial starting team is random, but known at the start of the strategy period.

Appendix II

System Setup

In our project setup we went for a multi-cloud setup which involved AWS and GCP. However, the same can be deployed in any cloud provider providing basic services.

Infrastructure prerequisites

Some of the pre-requisites of the entire project setup are:

1. UNIX based system(s) - are where most of the application code resides and few of the design concepts like using a CRON are most suited for UNIX.

2. A message buffer system - have been used to split different stages of our data pipeline. An equivalent Database transfer system can be made to work, but this would be the most efficient way of handling the projects requirements.

Note: The requirement of UNIX is not mandatory and can be substituted with other operating systems. However, this was the most suitable one.

Software prerequisites

The project uses the following software:

Application Code

1. Python - version 3.7.4
2. NodeJS - version 12.13.0

Database

1. FaunaDB Enterprise Edition - version 2.9.0

Application Code Setup

Our complete application code can be found on GitHub for deployment

Stage 1 - Fetching Match IDs for games

This step requires the deployment of the files `match_fetcher.py` and `unique_extractor.py`. The combination forms a tightly coupled system which needs to be deployed together as they have a file-based information hand-off system. There can be only one instance of this deployment as there is no reliable way to sequence the data given by Steam's

API, and having multiple deployment would result in an increase of duplicate data flowing into the system. Another point to note is that the `match_fetcher` would need an API key to consume data which cannot be reused by any other stage of the pipeline.

`Match_fetcher` writes data into the file in the data directory, which is then processed by `unique_extractor` in a batch process. The batch process is setup to run using the `crontab` functionality of UNIX. The file `make_batch.sh` has the necessary steps to set authentication to our message buffer etc, and call the `unique_extractor` function. Ideally, the cron is setup to run every minute using the following expression, but can be modified based on processing capacity.

```
* * * * * <project-directory>/data_dump/make_batch.sh
```

The deployments for this stage has been setup in a dedicated machine to be as de-coupled as possible as failure of this process would mean we start losing information on games during the downtime.

Stage 2 - Fetching Match Details for a Match ID

This stage of the data pipeline is implemented in the `detail_fetcher.py` file, each needing a Steam API Key to function. This can be deployed as a fleet of processes. The following factors need to be considered when deciding on the number:

1. Number of API keys available
2. Capacity of the database - since this stage stores raw data into the database for other stages to process, we need to consider the capacity and amount of traffic the database in handling. This is important as our pipeline is designed to be read intensive and a lot of processing has been moved to the write stage. Care needs to be taken not the flood the system with too many requests.

Note: While the deployments in the implementation were done on the same server due to budget constraints, having it on separate regions would improve the resilience of the system.

Stage 3 - Data Processing

The data processing stage is set up similar to the previous stage and is implemented in the `data_processor.py` file. and can be deployed in parallel. The difference here is that this stage is a completely internal process. Data is fetched in raw format from the database, processed, and persisted back.

The considerations on number of instances will be same as the previous step, however, the overall number can be less than the number of instances of the previous step as the previous step needs to have a wait period to fetch data from the API, and this stage has no such limitation.

Stage 4 - Data Analytics

The data analytics stage is contained with a simple node file, which can be run to return results to the business questions. The queries mentioned in this file are used comprehensively to power the visualization stage of the pipeline.

Stage 5 - Data Visualization

Client Application

The data visualization layer was created by setting up a web application. This application serves as the client side for all the analytics and was created by modifying a given react dashboard template. The real-time graphs were displayed by adding another key to the chart variables file called “realTimeData”, which takes dynamic array as input, and creates the variable in the form the graph would want.

The application was hosted on gh-pages by following step by step instructions on a medium post. The domain name used was “prabhjotrai.com”, with a CNAME of “dota”.

Server Side Application

The server side application was created using NodeJS and Express, by following step by step instructions on another medium post. Then, the application’s compilation was changed from native javascript compilation to ES6, by adding babel compilation tool as dev dependencies to the NodeJS application’s package.json. This is also illustrated step by step in another medium post. An interval function was created for fetching the match count per second and match mean and average time per five seconds and the results were pushed to a dynamic array which was sent over the Pusher to the client side.

Fauna DB Setup

The setup for the Fauna DB has been done by following the steps given in the official documentation which can be found here. A step which is not very intuitive in the documentation is the content of the faunadb config YAML.

A sample config yaml template is provided below:

```
---
auth_root_key: <secret-key>
cluster_name: <cluster-name>
network_listen_address: <private-ip-of-instance>
network_broadcast_address: <public-ip-of-instance>
```

A more detailed explanation of the different configuration options provided can be found here.

Note: The cluster name cannot be changed once initialized, and multiple replicas of the database must have the same cluster name.

Visualization Tool Setup

The visualization tool consists of setting up two applications, one is client facing React App and the other is the server side NodeJS application. In order to make changes to the dashboard, follow the following steps:

1. Clone the repository `git@github.com:raiprabh/dota-pipeline.git`.
2. Change directory into the `web` folder, by `cd dota-pipeline/web`.
3. Please ensure that `yarn` is installed in the server machine. If not, follow the steps on the yarn website.
4. Run `yarn install` to install the dependencies.
5. Run `yarn start` to run the application in development mode.
6. Run `yarn deploy` to publish the changes to the production dota.prabhjotrai.com. (this command internally calls `yarn build` to build the production code)

As mentioned before, the visualization tool is fed real-time data through a backend NodeJS application. This application can be setup on a server or on a local computer, and running the application will send the data to all the open instances of the frontend application (both in development and in production mode).

The following are the steps to setup the server side application:

1. Clone the repository `git@github.com:raiprabh/dota-pipeline.git`.
2. Change directory into the `server` folder, by `cd dota-pipeline/server`.
3. Please ensure that `yarn` is installed in the server machine. If not, follow the steps on the yarn website.
4. Run `yarn install` to install the dependencies.
5. Run `yarn start` to run the server.

Alerts and System Monitoring

A monitoring system has been put in place to check on the key points of failure of the system and has been implemented via the file `monitoring_alert.sh`. The monitoring functionality here mainly monitors the provenance information tracked with the data and sends out email alerts based on major deviations in the data.

This bash script makes use of the output of `alert_processor.js` file to perform the various validations.

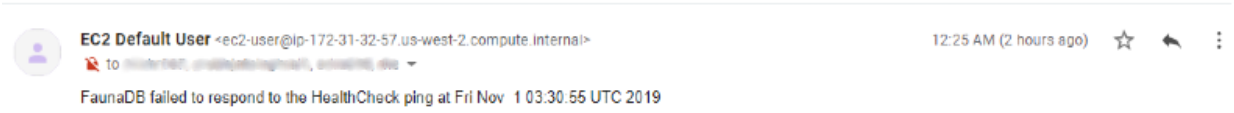
We make use of the crontab functionality to periodically check and notify the users in case of any issues which need their attention.

Crontab command

The following command can be used to initiate this functionality

```
*/15 * * * * sh <project-directory>/crons/monitoring_alert.sh
```

Sample Email of the alert



The content of the email will change based on the tracker which fails validation.

Note: This file will need to be modified with the email distribution which needs to receive the alerts, and the sending email address will have to be marked as **not spam**

Appendix III

FaunaDB Indexes

Note: The complete code for creation of all indexes can be found on GitHub.

Indexes are a powerful concept in FaunaDB and are required for any and all range queries. Like in SQL database, Indexes are created on collections. However, indexes in FaunaDB need to mention two fields, `terms` and `values`. The `terms` field denotes the fields in the document by which data can be queried, and contains fields which usually occur in the `WHERE` clause of SQL. The `values` field denotes the values that the index needs to return with queries, and contains fields which occur in the `SELECT` clause of SQL.

Project Indexes

The following are the indexes created on the database and the task they perform:

`Matches_raw_duration`

This is an index created on the `raw_matches` collection, which is the HiFi dump of data acquired. This index allows you to query all documents based on the value of the `duration` field inside the `result` document.

`Matches_raw_prov_api_duration`

This is an index created on the `raw_matches` collection. This index allows you to query all documents based on the value of the `apiCallDuration` field inside the `dataFetchStage` provenance record.

`Matches_raw_fb_time`

This is an index created on the `raw_matches` collection. This index allows you to query all documents based on the value of the `first_blood_time` field inside the `results` document.

`Match_by_ts`

This is an index created on the `matches` collection. This index allows you to query all documents based on the value of the `start_time` field of the document.

`Heroes_temporal_winrate`

This is an index created on the `heroes_temporal` collection. This index allows you to query all documents using the `id` and `win` field of the record, and returns the `match_start_time`, `id`, and `win` fields of the document.

`All_match_prediction`

This is an index created on the `match_prediction` collection, and has no terms or values. This index will only allow you to perform a select all on the collection.

Appendix IV

FaunaDB Clustering Basics[11]

FaunaDB uses a mesh-oriented approach to clustering: Deploy a node, point it to its peer, and go. The system takes care of everything else.

The key components of a FaunaDB cluster are:

1. Node
2. Replica
3. Shard/partition

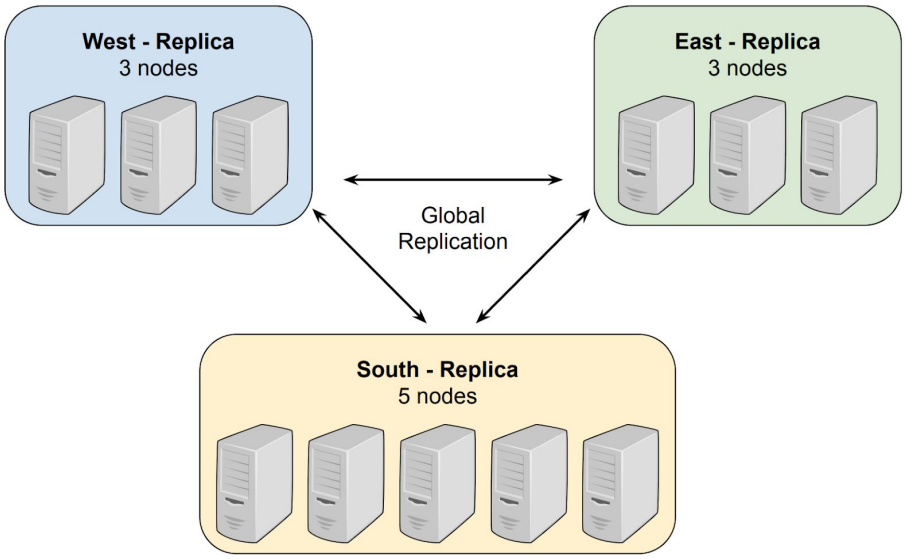


Figure - FaunaDB example cluster

Fauna's physical data layout can be described entirely by three simple concepts, node, replica, and cluster.

Node

A node is a single computer with its own IP address in which FaunaDB is installed and running. The Fauna nodes can run on all public cloud services, including Amazon, Azure, Bluemix, and Google, in addition to running on a private cloud, an on-premise solution, a virtual machine, a docker image, and several other platforms.

Replica

A replica is comprised of a group of one or more node(s). A node can belong only to a single replica. A fundamental property of a replica is that it contains a complete copy of the Fauna data, including end user data, system data, and the transaction log. Within a replica, a particular piece of data is normally found on exactly one node; there is no data redundancy provided by Fauna within a replica. Having multiple replicas, each containing the full set of data, is what provides redundancy in a Fauna database cluster. Another common term for a replica would be a datacenter.

Cluster

A FaunaDB cluster is the highest entity. Under the cluster is both a physical layout and logical layout. To review, the physical layout is comprised of individual nodes. Every node is a computer with a unique IP address. Nodes are grouped into replicas, with every node belonging to exactly one replica. The main significance for this grouping is that a replica contains a full copy of the data and can vary in size. Within a replica, a particular piece of data is normally found on exactly one node; there is no data redundancy within a replica.

The FaunaDB Cluster has a logical data model comprised of databases, collections, and documents. This data model is a semi-structured, schema-free object-relational data model, a strict superset of the relational, document, object-oriented, and graph paradigms. The model starts with the database which can be laid out in traditional flat fashion, in a hierarchical or a combination of flat and hierarchical as seen in Figure 2.

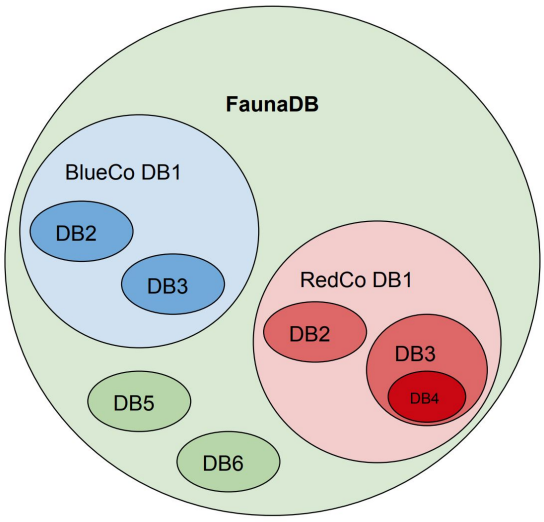


Figure - Database which can be laid out in traditional flat fashion, in a hierarchical or a combination of flat and hierarchical

References

[1] https://en.wikipedia.org/wiki/Dota_2

[2] https://dota2.gamepedia.com/Abilities

[3] https://docs.opendota.com

[4] Steam Community Documentation: https://steamcommunity.com/dev/apiterms

[5] FAQ section under https://dev.dota2.com/showthread.php?t=58317

[6] https://dbdb.io/db/faunadb

[7] https://docs.fauna.com/fauna/current/introduction

[8] https://fauna.com/faunadb

[9] Full JSON - https://jsonblob.com/c65dd64c-e63e-11e9-8b82-0b0ad04e74ce

[10] Full JSON - https://jsonblob.com/f44f33d2-e63f-11e9-8b82-dd807557ebd2

[11] https://fauna.com/blog/introduction-to-faunadb-clusters

[12] https://dota2.gamepedia.com/Matchmaking

[13] https://dota2.gamepedia.com/Game_modes#Captains_Mode

[14] https://dota2.gamepedia.com/Game_modes#All_Pick

[15] https://dev.dota2.com/showthread.php?t=58317