Video Game Reviews And Ratings And Their Affect On Sales

Joshua Vach

I. Project Overview

This big data system combines several databases of video games looking at reviews, age ratings, and sales.

A. Domain

A big issue that has been plaguing the video game industry is large game studios creating games that people do not like or want to buy. Many studios see one game do well and try to copy it without looking at broader trends or customer reviews.

B. Problem Statement

This system will allow for the quick answering of several questions related to how well video games perform using Metacritic ratings, along with age ratings and total sales. As well as taking budgets for some of the most expensive games into account as well.

C. Scope

This system will create a usable and query able system that allows for quick access of several key points from multiple large databases containing video game reviews and sales and another data set containing development costs for several of the most expensive games.

II. System Overview

A. Data Sources

I have 3 sources of data 2 are CSV files from Kaggle about video game sales and the other holding metacritic reviews for video games. The other file is an XLS file containing budget calculations for several of the most expensive games.

B. Data Characteristics

- **Volume:** The 3 datasets vary wildly in size the largest being the metacritic dataset with 512 thousand rows and 14 columns. The second largest is the Video game sales data set with 11493 rows and 11 columns inside its data set. The smallest is the budget XLS file with only 15 rows and 8 columns.
- **Variety:** There are 3 different structures in my data sets
  - **Metacritic:** CSV file with the table structure being: title, poster, release date, developer, genre(multiple data points), platforms(multiple data points), product rating, overall metascore, overall user rating, reviewer name, reviewer type, rating given by reviewer, review date, and review text.
  - **Sales:** CSV file with the table structure being: rank, name, platform(single data point), year, genre(single data point), publisher, NA sales(in millions of units), EU sales(in millions of units), JP sales(in millions of units), other sales(in millions of units), and global sales(in millions of units).
  - **Budget:** XLS file with the table structure being: characteristic, Name(year), developer, publisher, development cost(in millions USD), marketing cost(in millions USD), total cost(in millions USD), and total cost with 2023 inflation(in millions USD).

C. Stack Layers

This project will engage with multiple layers of the big data stack. the first one being the syntax as having two different file types makes it so that syntax must be changed and made so that it can work between different file types. The second layer that will be engaged is the data models as all of the different data sources have very different ways of storing their data so the data between the 3 must be normalized to make sure their data frames do not cause errors with the others. The third layer would be data stores as there needs to be a way to store all of the data being processed the easiest way would be to use a wide column model to store the different schemas before more normalization can be done. The fourth will be processing as the system will use MapReduce to take various pieces of data from all over the new dataset and use it to calculate various queries like total sales per million dollars spent or average sales per metacritic score. The fifth will be querying as the purpose of this system is to be able to pull data from all these sources simply and easily using various partitioning strategies. The system will also touch upon the storage layer by using HDFS storage system.

D. Assumptions

The biggest simplification being made is with regards to the metacritic database the reviews and posters are not necessary for the system so will be removed to save on space which will allow for that data set to be shrunk significantly due to many rows being for the same game.

III. Implementation Approach

A. Technology Choices

The system will use several different technologies in order to store and process data efficiently. The first technology would be using a wide-column model, in this case Apache HBase, to store all of the different variations of data structures in the data sources. The second will be a MapReduce architecture that takes the data from the wide-column model and creates

usable data from the key which in this case would be the name of the game. The third will be SparkSQL to use for querying for our data after its processing. The fourth will be a HDFS system for storage of data.

B. Processing Model

The systems processing model will be batch processing because there is a lot of data that is pre-collected which can be worked through on a schedule efficiently without any need to add new data over time.

C. Scalability Plan

Due to the nature of my input data sets scalability should not be a problem since they are all fixed sets of data, but if scalability was needed storing the new data in the wide-column model and then continuing on with batch processing will be enough to handle any increase in data needing to be processes.

D. Metrics

The metrics that this system will use will be throughput. Since there is a finite amount of data determining how fast it computes parts is for more useful then determining how much has been computed.

IV. Core Concepts and Technologies

A. HDFS

- **Source:** https://pages.cs.wisc.edu/~akella/CS838/F15/838-CloudPapers/hdfs.pdf

[1] This source explains how a Hadloop Distributed Files System or HDFS works alongside stating what can work with it. The HDFS works by splitting data into nodes that are replicated 3 times and attached to a name node that allows easy access. The client allows for creating new blocks and writing data onto those nodes as well as access to metadata and a commit log. This applies to the system being created because the data being used for the system needs to be stored in a way that is both easily accessible and modifiable while also being able to scale well and having backups for the data in case something goes wrong. The portions that will be adapted into the system are the client, name node data node, and checkpoint node. The client will allow for adding the required data sets into the HDFS system which consists of the name nodes and data nodes. The checkpoint node is not necessary but is useful when it comes to having more documentation about the data stored in the HDFS. HDFS will mainly be used to store the HBase wide-column model used for storing the various data schemas.

B. Apache HBase

- **Source:** https://hbase.apache.org/book.html#datamodel

[2] Apache HBase consists of row keys connected to various column families that are not necessary for every row to have. Column families that are not present in a row do not have space allocated to them. HBase also stores multiple versions of each cell with the same row for data redundancy. All column families need to be specified at table creation but can be added with .addColumn or changed with .modifyColumn. MapReduce does not have innate access to an Hbase database so it will be necessary to add dependencies onto the HBase CLASSPATH in order to allow MapReduce to access the data. Basic data model operations like get put and scans are also present inside of HBase which allow for reading from a location, adding new data or updating data, and allowing multiple rows to be iterated through. This data store is necessary for the system to function as the data from the three different sources have some overlapping qualities but also have different data so it is necessary to have the ability to have different columns for those differences. The system uses for HBase will be one large table containing all of the data from our source databases after they have been trimmed of unnecessary data in order to save on space. This table will have the column families relating to the game name, sales, reviews, age rating, genre, publisher, developer, platform, and budget.

C. MapReduce

- **Source:** https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[3] MapReduce splits the input data set into independent chunks of data that are then processed by the map task that run in parallel with the other map tasks. The maps are then sorted by a framework and put into a reduce task. The framework also handles re-execution of failed tasks. Using HDFS the compute nodes of MapReduce are the same as the storage nodes used inside of the HDFS. The framework of MapReduce contains one ResourceManager, one NodeManager per node, and an MRAppMaster. Input/Output locations are determined by the application and along with other parameters make up the job configuration. The job client itself takes this and sends it to the ResourceManager which then distributes it to the NodeMangagers. This applies for our system because there is a large amount of data in multiple different storage rows with different column values with the same key value(typically game name). This systems configuration will take the data from these identical keys and use them to calculate various useful pieces of data like average sale per metacritic rating, sales per millions of dollars of budget, or average sales per genre.

D. SparkSQL

- **Source:** https://spark.apache.org/docs/latest/sql-programming-guide.html

[4] SparkSQL is created specifically for data processing and has interfaces that provide more specific information about the data and the process that is computing said data. When using a different programming language with SparkSQL the output will be returned as dataframe. SparkSQL is relevant to this system because there will be lots of taking data in and out of databases and having a specific language specialized in that will make things easier in the long term. The uses in the system will mainly be modifying some data to make it cleaner

by removing unnecessary data and then transferring the data into the HBase wide-column model and also putting the output of the MapReduce into its own database and accessing its data with queries.

E. Data Partitioning

- **Source:** https://learn.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning

[5] Data Partitioning into separate data sets based on horizontal, vertical, or functional partitioning can help make a system not only run faster but allows for data to be easier to access and allows for an easy upscale of the system if needed. Horizontal divides the dataset into different datasets with the same schema divided by subsets of the same data. Vertical divides the dataset by having separate datasets for different columns of data based on various criteria including how frequently the data is accessed. Functional partitioning separates read-only and read-write data. This is relevant to the system because the output data for the MapReduce will have several different types of data that are different from each other and could be separated to help with efficiency. The system itself will use vertical partitioning for its output code separating the datasets that used different keys in the MapReduce.

V. References

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler.
[2] A. H. Team.
[3] ApacheHadoop.
[4] ApacheSpark.
[5] claytonsiemens77, "Data partitioning guidance - azure architecture center."

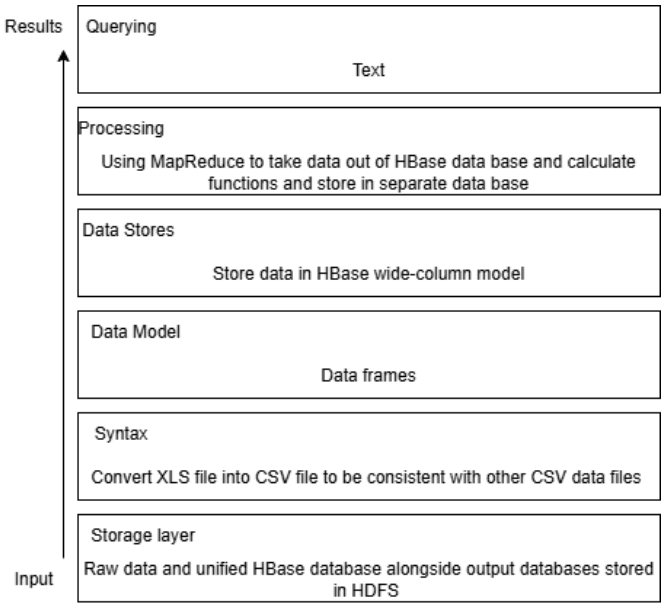

Fig. 1. Stack Architecture Diagram

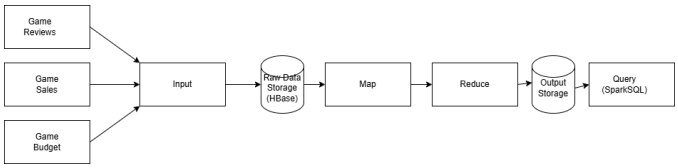

Fig. 2. Data Flow Diagram