**Introduction**

In the process of making a game to be consumed by millions, one of the many problems developers run into is regarding data storage and retrieval. Databases soundly resolve the problems faced by these developers. However, not all databases are made equal: while all databases have their pros and cons, only the most relevant database will be considered for our mobile game. To this end, a database’s most important operation is that it must not only perform extremely fast reads of the game state when requested by the player, but also process game state updates and game event inserts. This database must be able to handle player queries near-instantaneously, such as showing the top 10 players within a region. A database must be able to periodically execute queries to determine players that have high activity (or lots of events) in a given time. This paper will assess three databases – Cassandra, CosmosDB, and Neo4J – as well as their pros and cons, and how they can be relevant towards satisfying – or hampering – the fulfillment of these conditions. To this end, one of the three databases will be chosen on the breadth by which they fulfill the imposed conditions.

**Database #1, “Cassandra”**

The first database to be assessed is Cassandra, a NoSQL database. Since Cassandra is masterless, it is highly scalable: doubling capacity is possible by double the nodes, allowing it to easily scale under high stress. Moreover, Cassandra is able to scale back as easily as it can scale up. Cassandra also does not require more CPU power, RAM or a faster disk as data volume increases, saving a significant amount of money for a game development company. Additionally, Cassandra offers a two-sided coin when it comes to designing schema. If one is able to design a good schema which covers a wide range of queries, Cassandra could potentially reduce a lot of work. Cassandra also has a high fault tolerance due to its data replication structure, allowing for high loads as player count increases. Another benefit of Cassandra is that it is capable of reading CSV files. If player information is kept in CSV format, using Cassandra allows for efficient reading of necessarily data.

However, Cassandra comes with its drawbacks. Cassandra’s data can only order by the original table and not by the aggregated function – this excludes sums, count or any columns not part of the original table. Moreover, the other side of the schema coin is that if a schema is poorly designed – that is, it does not cover enough possible queries – then it can lead to massive inefficiencies. For example, for different queries, one might have to upload the same dataset multiple times: one for each query, since it requires a specific structure. It is also encouraged to denormalize over normalize data. When a game event happens, the previous game states of players are usually not updated. Instead, they are saved, and a new state is created. If this occurs every time, this process can result in a massive amount of unoptimized data. Finally, while Cassandra was originally optimized for fast writes, the tradeoff was slower reading operations. This is problematic since, while new data can be easily introduced to the database, retrieving information tends to significantly slower. This is an issue when the player wants near real-time updates for their states (such as where they are in terms of their current rankings, or when they search up the top 10 players of a region).

**Database #2, ‘CosmosDB’**

CosmosDB is another NoSQL database. However, unlike Cassandra, CosmosDB is flexible: it can support multiple APIs, such as SQL and MongoDB. Since it can use SQL as its query language, this means that it can easily perform order-by operations for in-game events and join tables which were not in the original table in question – something Cassandra is unable to do. CosmosDB’s functionality also allows for the option to choose between five levels of consistency, which can balance, or shift priority between, consistency and speed. This variable scale of options allows for more flexibility to match different circumstances or scenarios as they arise. Moreover, CosmosDB is optimized for write-heavy workloads. This is useful for our mobile game because of the massive amounts of data which will eventually need to be processed as the game grows in both content and players. CosmosDB is also efficient at handling time stamps. This database can also reliably handle requests during peak usage. This reliability is beneficial for our game because there may be times at which the game experiences heavy player loads – such as at launch or during in-game limited time events.

Like Cassandra, CosmosDB faces drawbacks. It most efficiently reads data in JSON format: this could be cause for inefficiency in data format conversion (from CSV, for example). Furthermore, every time an interaction is made between the game and the player, a row is generated as a game event. If there are millions of players, this means millions of rows will be generated. To work on data, it must first be uploaded. However, CosmosDB has throughput limits, which limits aspects of the database, such as volume. If the size of the data one attempts to upload exceeds the throughout limit, it is unable to be uploaded. CosmosDB also has an internet speed requirement, with larger data sets needing more request units (RU) to perform a read and write operation. RU is a rate-based currency, and is dependent on the size of the item, with bigger items requiring more RU per read. There is also the problem of properties per item: the costs to run operations on an item increases as the properties of that item increase. Costs further increase in RU with the complexity of queries and server-side scripts. If the throughput limits are too low to sufficiently keep up with the increasing player count, player queries (such as searching for the top 10 players in a region) could face delays, falling short of the expected ‘near real-time’ requirement of querying. This throughput limit can be expanded by purchasing packages. Being compatible with various APIs, CosmosDB is a highly flexible database system that can scale instantly with the kinds of data loads it is presented with. However, while CosmosDB *can* be capable of instantaneous, high-speed service, its performance is also dependent on (and potentially hampered by) how deep one’s wallet is: fees such as CosmosDB’s throughput limits could cut into profits and future development funds (and developmental timeline) of a mobile game.

**Database #3, “Neo4J”**

Neo4J is an open source, NoSQL, native graph database that provides an ACID-compliant transactional backend for applications that has been publicly available since 2007. Its graphs represent entities as nodes and the edges between the nodes represent the relationships between the entities. Neo4J uses a pattern-based graph query language called Cypher to store and retrieve data from the graph database. Using Neo4J can be beneficial as it is incredibly fast and efficient at reading data. For example, it can perform real-time operations like resolving inherited data in a file tree at high speeds. Also, unlike Cassandra, Neo4J can perform ‘order-by’ operations – this is useful when players attempt to order the top 10 players in the region, for example. Neo4J is also schema-free: this means that the data model can adapt and change, and can easily import CSV files using Cypher to transform content into graph structures. As before, should player information exist in CSV format, having a database which can effectively read it makes it beneficial towards overall performance efficiency. Moreover, Neo4J is highly scalable. When one increases the data, the database scales accordingly: as players increase – and the number of game states similarly increases – the amount of data will expand. A database must be able to handle this increasing data load effectively, and Neo4J’s ability to scale in proportion to the data presented to it makes an effective database system to address possible concerns which a game developer might face.

However, no system is perfect: Neo4J also faces drawbacks. There is no support for sophisticated index mechanisms, and for our mobile game, one of the conditions to be met is the ability to search within ranges – e.g., time ranges. Also, while Neo4J is ‘large read-volume’, the drawback to this is its ‘low write-volume’. In other words, in cases with high write-volume, writing is the bottleneck for Neo4J. This is relevant for our game because as player count increases, and the uploading of write-volume increases, the system becomes more and more inefficient. Finally, graphs are meant to model relations between entities. In our game, the entities will be players with an id, timestamp, power, gold, level, and so forth. While a player will relate to their own game state, they will not often relate to other players’ game state, and the different players’ game states are not related, either. Since part of the functionality of graph databases like Neo4J are to measure and draw relationships between entities, this aspect of Neo4J would not be used to maximum efficacy.

**Best Database System**

After analysis of relevant factors, CosmosDB appears to be the best database of the three. Part of the game’s requirement is that the database must be able to search within a range, which Neo4J cannot perform. Another requirement is to be able to order by count, which requires relating different tables together – both operations Cassandra cannot perform. Since CosmosDB is so flexible in its query language usage, it can satisfy all the requirements through use of SQL, for example. CosmosDB also satisfies the condition of handling time stamps with great efficiency and efficacy, which was another requirement of the database. Furthermore, the size of the data set provided is not so massive such that the throughput limit would have a substantial hampering effect, if any. Ultimately, while each assessed database has its pros and cons, CosmosDB fulfills the most requirements for our mobile game in question.