Micheal Willard
932480626
CS 496
Assignment 3 Part 1: Non-Relational Database Design

Data being Modeled

The purpose of this application is to create a team management system. The backend APIs will provide an interface for the system. This team management systems critical function is to inform a manager of what players are attending or not attending each game. To do this, the system must consist of games with game details and players with player details. Those are the primary groups of data. So those allow the data to be broken into two obvious entity groups, as will be discussed in the data modeling plan.

The game objects will each have the same consistent features, time, location, opponent and lists for who is attending or not attending. Application logic could be applied to determine who hasn't responded yet. The player objects will contain the data critical for this application which is the player's name and contact info consisting of an email address and phone number.

It's possible more information could be added to each object model, but the basic structure will remain as described.

Data Modeling Plan

The data will be modeled using Google App Engine's Python Non-Relational Database, also known as Cloud Datastore. It is a simple straightforward NDB with its own entity-based structure. There is exceptional documentation on the proper usage and implementation of it's features and syntax.

From Google's documentation, the Cloud Datastore relates to traditional databases as described in this chart:

Concept	Cloud Datastore	Relational database
Category of object	Kind	Table
One object	Entity	Row
Individual data for an object	Property	Field
Unique ID for an object	Key	Primary key

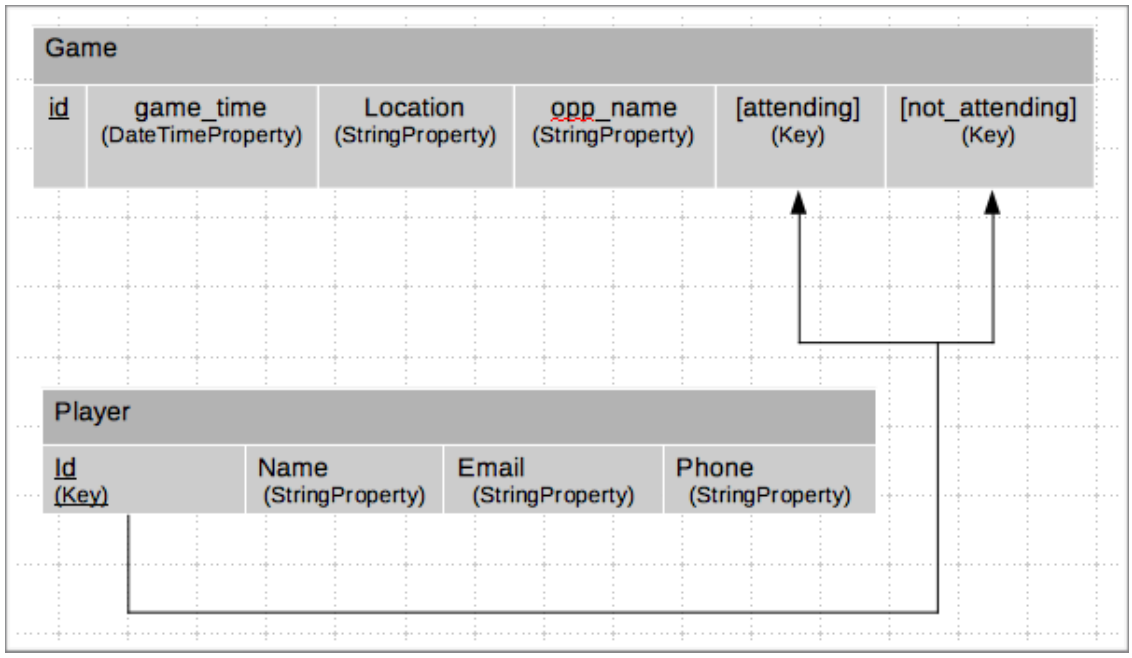
Uses cases of the data enclosed in the database will be weighted mostly towards writing. The game objects themselves won't be modified often, in terms of date, location and opponent once they have been initially input into the system. However, the players attending, not-attending the games will be affected at least once per player per game. Reading of the games happens frequently, but this is at a higher "view all" type level.

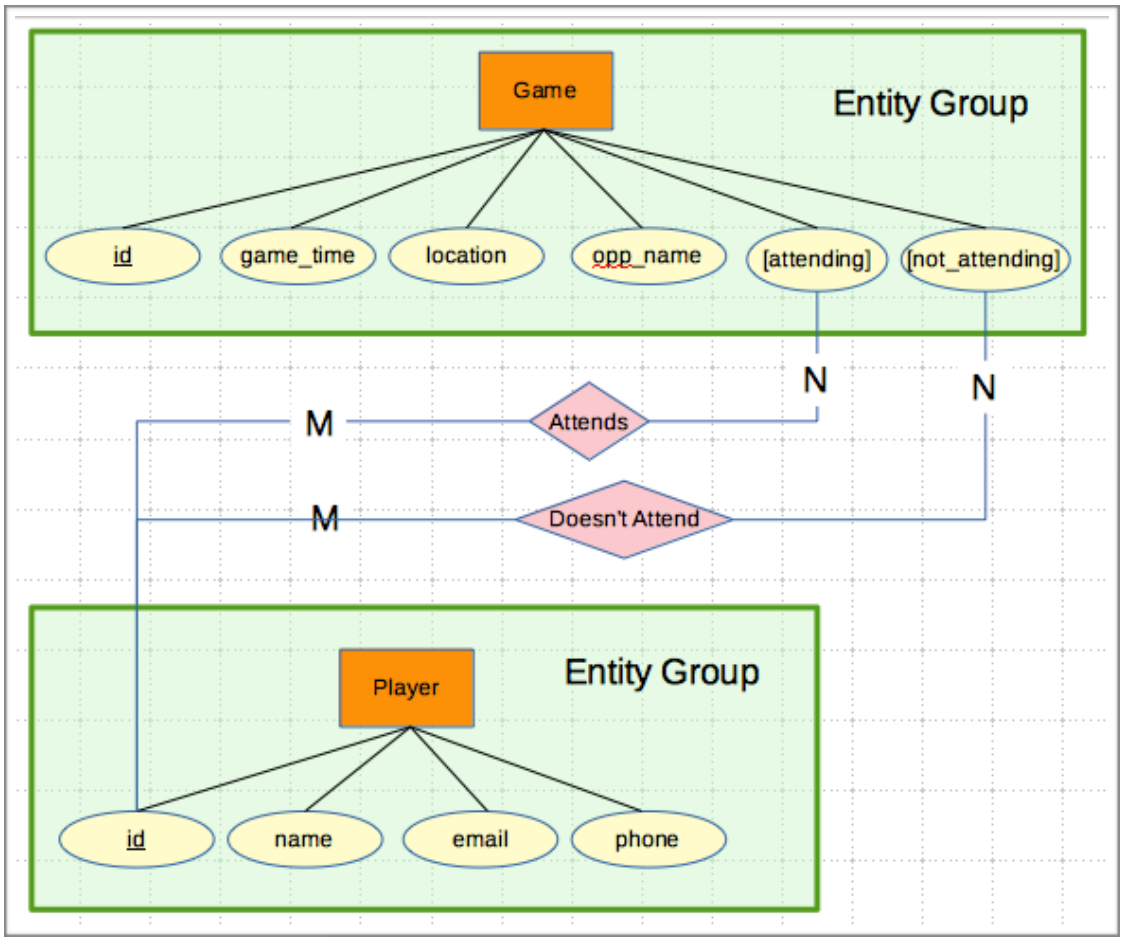
The data store plan should be able to maintain consistency, since each game is an entity group. When a game is accessed, a user will be able to see who is attending, not attending and

not responded to that game. If the user makes a change, they are still reading and writing within that entity group.

In this instance of the database, it is being used to control only data for one team. GAE's NDB allows for approximately 1 write per second. Since this instance is only for one team, it is unlikely any fault will occur with two players updating their status at the same time, and a manager trying to view that data at the same time. However, scalability must be taken into account. If this schema were to be scaled to handle multiple teams, like some existing applications out there, the breakdown of the entity groups allows the data to be consistent. If each game is an entity group, you still have a small likelihood of being negatively impacted by the 1write per second constraint.

Schema Diagrams





Research on Other Non-Relational Database

The other non-relational database that was considered for this implementation is Amazon Web Services' SimpleDB. Some of the potential issues for SimpleDB that were identified revolved around consistency and speed. SimpleDB has relaxed consistency restraints that can affect operations at the transaction level. There may be disagreements with what is read from the data right after a write to the data.

From the documentation for SimpleDB, it can mostly be described as AP without C (from the CAP theorem). The documentation describes it as highly available. Stored data items have multiple copies created automatically. Obviously that can cause the aforementioned issues with consistency. The way the data is being modeled would have to change to work around these consistency issues.

Additionally, the API support for SimpleDB isn't as strong and available as it is for Cloud Datastore's Python API support. This can make for a number of challenges in writing the actual code to access the data. It is designed to be used with other AWS services.

An additional con to the AWS system would be it's portability to other systems. The data stores done have a compatible format, and the code base would have to change dramatically. However, that issue could be potentially mitigated due to the fact that SimpleDB is very scalable. So the need to move to another system might not ever be a concern.

Google App Engine's NDB is consistent when dealing with single entity groups. When cross entity group queries are happening, the data is not necessarily guaranteed to be consistent. However the data modeling plan involves working within entity groups, where the consistency is critical. Google Cloud store is also highly available. Thus allowing it to be CA without P, which is the best option for this data and data usage.

SimpleDB attributes are typeless, whereas Could Datastore attributes can have types. This allows for more control and execution as an object-oriented system.