# Google App Engine High Replication Datastore

Google App Engine's High Replication Data Store is at the heart c nearly all web applications. **This document will broaden your understanding of what the data store is, how it replicates data to deliver content to your users, and how the data store can be effectively used in order to avoid replication lag.** At the end of this document we include a list of useful vocabulary and additional reading to help further your understanding of the data store.



### *The High Replication Data Store*

Google App Engine has its own built in API that is used for storing data which is great for us because it means we don't have to write our own datastore methods. The High Replication Datastore (HRD) allows us to replicate data across multiple datacenters. Objects known as entities are written to the datastore and each entity has a uniquely defined key In the building a basic blog solution, Steve points out a method that h uses to build blog keys, which is essentially used to build a parent-child relationship between the blog and its posts. Entities are not required to have parents but it is often easier to query the data store by paren nodes (called ancestors) as opposed to querying each entity.

### *Conceptualizing the Data Store*

An easy way to visualize this is picture the datastore much like your OS's file system structure. Ancestors are similar to your folders, and it's children (called descendants) are similar to sub-folders or files. In Steve's setup of the blog we have an ancestor node named 'default' (o any name we choose) and it's children are the set of blog entities which have an assigned key which links them to their ancestor. Entities tha share a common parent are known as an entity group. This entity group also includes the ancestor of the group.

### *Consistency in Results*

One issue with using the HRD is the delay between the time of a write

being committed and it's visibility in the datastore. In some instances you may not immediately see the previously entered piece of data, i worst case instances it can take up to 10 minutes for a write to be visible across all datacenters. As such queries across multiple entity groups can only guarantee eventually consistent results. That is queries to the datastore may not reflect the most recent changes made to the data. To obtain strongly consistent results we need to use ancestor queries to limit our results to a single entity group.

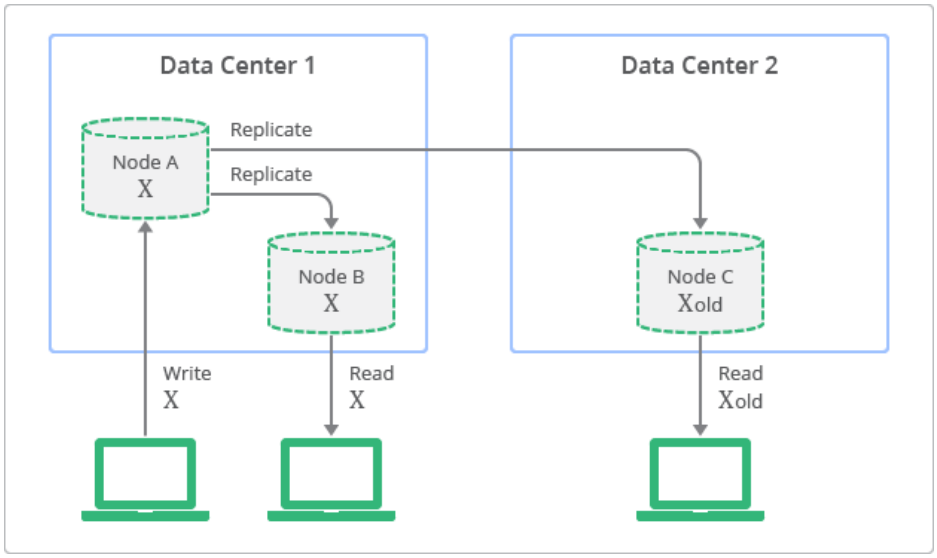


*Fig 1 : Replication with Eventually Consistent Data*

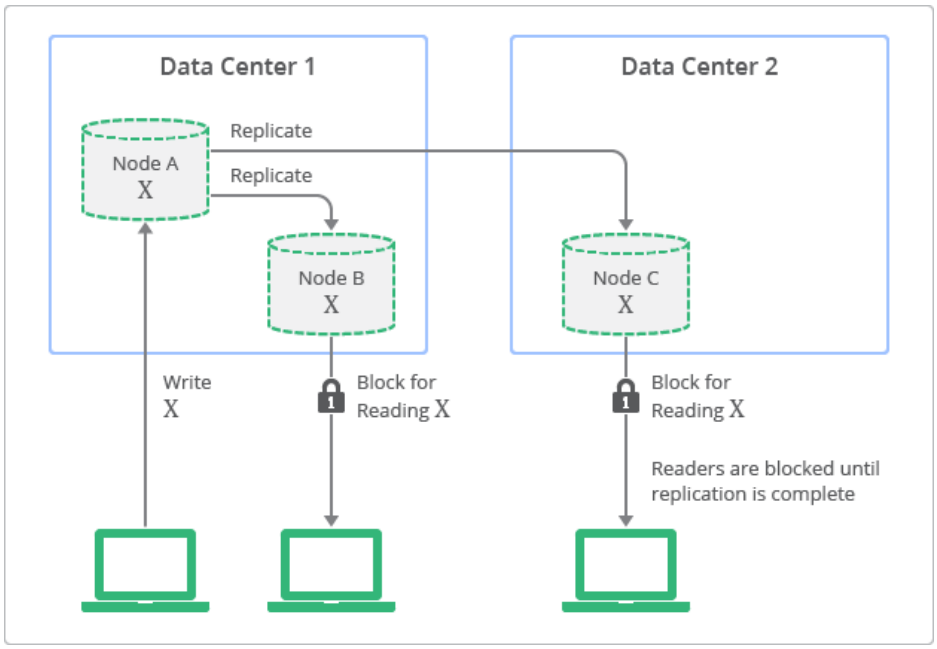


*Fig 2: Replication with Strongly Consistent Data*

For, example imagine we had multiple blogs hosted on our site. If ou data is structured in a way where each blog post is its own root entity i.e. it has no ancestor, when we query our database the datastore would automatically search across all entities, and would return results regardless of whether or not the data is up to date. If instead we organized our data so that each blog has its own unique ancestor ID and each blog post has a key associated with its ancestor, then when we query the database we will get strongly consistent results. This is because data operations are applied to the whole entity group and th ancestor query will not return results from a group until it is entirely up to date. Between 4:19 and 7:00 of the Registration Solution in problem se 4, Steve explains in detail methods for organizing the way data is stored into the datastore to insure strongly consistent results.

***Vocabulary and Additional Topics:***

Ancestor: Entities which other entities reference. Another name for a parent node.

Ancestor Query: A query which only retrieves entities who have descended from a specific ancestor

Descendant: Children of an ancestor entity. The ancestor of a descendant is defined by a unique key.

Entity: Data Objects stored in the Datastore. Entities may or may not have a parent entity. Entities with no parents are known as Roo Entities.

Entity Group: Entities that share a common ancestor form an entity group, this group includes the ancestor.

Eventually Consistent : The guarantee that if no updates are made to the data that eventually all accesses to that item will return the las updated value. Eventually consistent data may not accurately reflect the most recent writes to the data due to replication lag.

Filter: Set of constraints chosen by the user to limit which entities are retrieved from the datastore.

High Replication Datastore (HRD): The primary data repository used by Google App Engine.

Key: An identifier used to associate descendants with an ancestor entity.

Kind: The type of the entity stored in the database (i.e. integer, String

object)

Paxos Algorithm: The underlying algorithm which makes the HRD possible

Query: Method for retrieving entities from the datastore

Replication Lag: The time between when data is written to the datastore and when the data is available to be read.

Strongly Consistent : All data is accessible when and does not suffer from the drawback of replication lag

Transactions: A datastore operation across one or more entities

Structuring Data for Strong Consistency, Balancing Eventual Consistency with Strong Consistency