Note: This page describes system behavior for Datastore databases that have not yet upgraded to Firestore in Datastore mode.

Firestore (/firestore/) is the new version of Datastore and removes several Datastore limitations (/datastore/docs/firestore-or-datastore#in_datastore_mode).

This document discusses achieving strong consistency for a positive user experience, while embracing Datastore's eventual consistency model for handling large quantities of data and users.

This document is intended for software architects and engineers wanting to build solutions on Datastore. To help readers who are more familiar with relational databases than non-relational systems like Datastore, this document points out analogous concepts in relational databases. The document assumes that you have a basic familiarity with Datastore. The easiest way to get started with Datastore is in Google App Engine using one of the supported languages (/datastore/docs/reference/libraries). If you have not yet used App Engine, we suggest you first read the Getting Started Guide (/appengine/) and the Storing Data (/appengine/docs/python/storage) section for one of those languages. Though Python is used for example code fragments, no Python expertise is required in order to follow along with this document.

Note: The code snippets in this article use the Python DB Client Library for Datastore, which is no longer recommended. Developers building new applications are **strongly encouraged** to use the NDB Client Library (/appengine/docs/standard/python/ndb), which has several benefits compared to this client library, such as automatic entity caching via the Memcache API. If you are currently using the older DB Client Library, read the DB to NDB Migration Guide (/appengine/docs/standard/python/ndb/db_to_ndb)

Non-relational databases, also known as NoSQL databases, have emerged in recent years as an alternative to relational databases. Datastore is one of the most widely used non-relational databases in the industry. In 2013 Datastore processed 4.5 trillion transactions per month (Google Cloud Platform blog post (http://googlecloudplatform.blogspot.com/2013/05/reducing-app-engine-datastore-pricing-by-up-to-25-percent.html)). It provides a simplified way for developers to store and access data. The flexible schema maps naturally to object-oriented and scripting languages. Datastore also provides a number of features that relational databases are not optimally suited to provide, including high-performance at a very large scale and high-reliability.

To developers more accustomed to relational databases, it may be challenging to design a system that leverages non-relational databases, as some characteristics and practices of non-relational databases may be relatively unfamiliar to them. Although the Datastore programming model is simple, it is important to be aware of these characteristics. Eventual consistency is one of these characteristics and programming for eventual consistency is the main subject of this document.

Eventual consistency is a theoretical guarantee that, provided no new updates to an entity are made, all reads of the entity will eventually return the last updated value. The Internet Domain Name System (DNS) is a well-known example of a system with an eventual consistency model. DNS servers do not necessarily reflect the latest values but, rather, the values are cached and

replicated across many directories over the Internet. It takes a certain amount of time to replicate modified values to all DNS clients and servers. However, the DNS system is a very successful system that has become one of the foundations of the Internet. It is highly available and has proven to be extremely scalable, enabling name lookups to over a hundred million devices across the entire Internet.

Figure 1 illustrates the concept of replication with eventual consistency. The diagram illustrates that although replicas are always available to read, some replicas may be inconsistent with the latest write on the originating node, at a particular moment in time. In the diagram, Node A is the originating node and nodes B and C are the replicas.

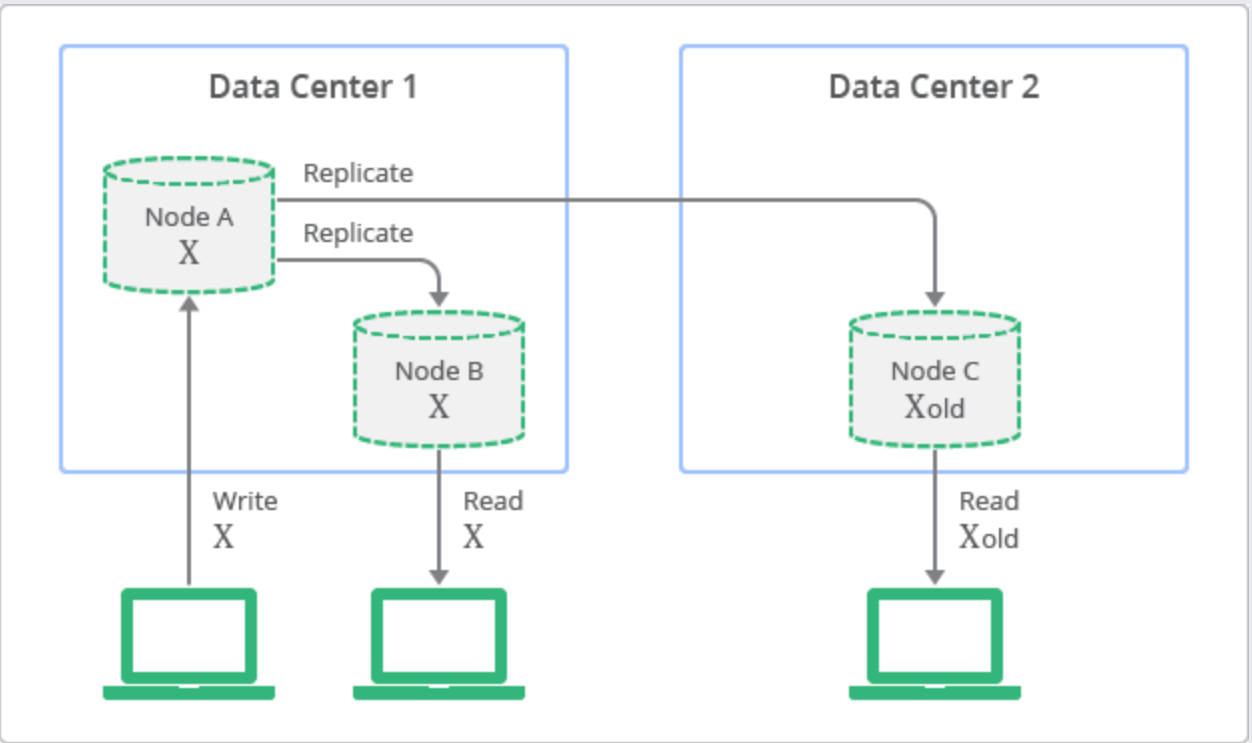

Figure 1: Conceptual Depiction of Replication with Eventual Consistency

In contrast, traditional relational databases have been designed based on the concept of strong consistency, also called immediate consistency. This means that data viewed immediately after an update will be consistent for all observers of the entity. This characteristic has been a fundamental assumption for many developers who use relational databases. However, to have strong consistency, developers must compromise on the scalability and performance of their application. Simply put, data has to be locked during the period of update or replication process to ensure that no other processes are updating the same data.

A conceptual view of the deployment topology and replication process with strong consistency is shown in Figure 2. In this diagram, you can see how replicas always have values consistent

with the originating node, but are not accessible until the update finishes.

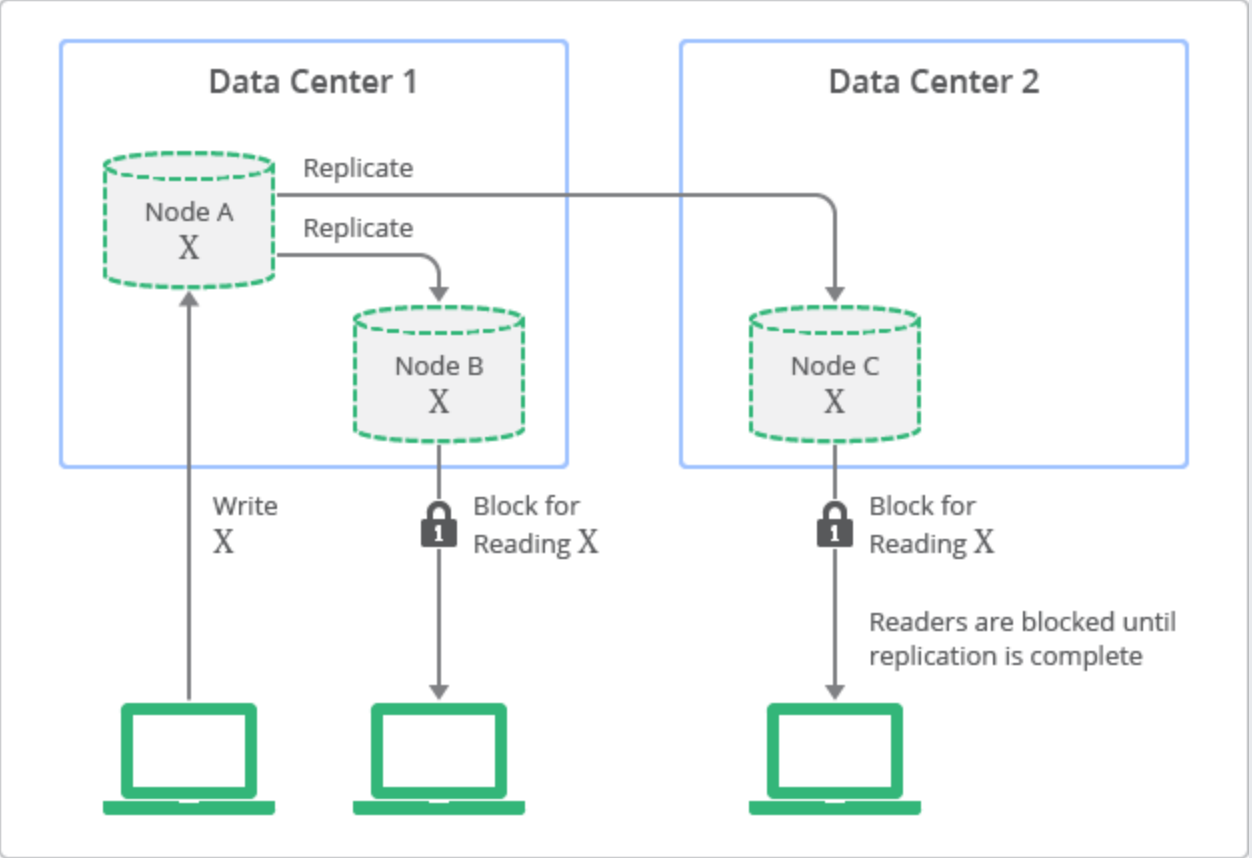

Figure 2: Conceptual Depiction of Replication with Strong Consistency

Non-relational databases have become popular recently, especially for web applications that require high-scalability and performance with high-availability. Non-relational databases let developers choose an optimal balance between strong consistency and eventual consistency for each application. This allows developers to combine the benefits of both worlds. For example, information such as “knowing who in your buddy list is online at given time” or “knowing how many users have +1’d your post” are use cases where strong consistency is not required. Scalability and performance can be provided for these use cases by leveraging eventual consistency. Use cases which require strong consistency include information such as “whether or not a user finished the billing process” or “the number of points a game player earned during a battle session”.

To generalize the examples just given, use cases with very large numbers of entities often suggest that eventual consistency is the best model. If there are a very large number of results in a query, then the user experience may not be affected by the inclusion or exclusion of specific

entities. On the other hand, use cases with a small number of entities and a narrow context suggest that strong consistency is required. The user experience will be affected because the context will make users aware of which entities should be included or excluded.

For these reasons, it is important for developers to understand the non-relational characteristics of Datastore. The following sections discuss how eventual consistency and strong consistency models can be combined to build a scalable, highly available, and highly performing application. In doing so, consistency requirements for a positive user experience will still be satisfied.

The correct API must be selected when a strongly consistent view of data is required. The different varieties of Datastore query APIs and their corresponding consistency models are shown in Table 1.

(#) (#)

Datastore API	Read of entity value	Read of index
Global Query (/appengine/docs/python/datastore/queries)	Eventual consistency	Eventual consistency
Keys-only Global Query (/appengine/docs/python/datastore/queries#keys-only_queries)	N/A	Eventual consistency
Ancestor Query (/appengine/docs/python/datastore/queries#ancestor_queries)	Strong consistency	Strong consistency
Lookup by key (/appengine/docs/python/datastore/entities#Python_Retrieving_an_entity) (get())	Strong consistency	N/A

Table 1: Datastore queries/get calls and possible consistency behaviors

Datastore queries without an ancestor are known as global queries and are designed to work with an eventual consistency model. This does not guarantee strong consistency. A keys-only global query is a global query that returns only the keys of entities matching the query, not the

attribute values of the entities. An ancestor query scopes the query based on an ancestor entity. The following sections cover each consistency behavior in more detail.

With the exception of ancestor queries, an updated entity value may not be immediately visible when executing a query. To understand the impact of eventual consistency when reading entity values, consider a scenario where an entity, Player, has a property, Score. Consider, for example, that the initial Score has a value of 100. After some time, the Score value is updated to 200. If a global query is executed and includes the same Player entity in the result, it is possible that the value of the property Score of the returned entity might appear unchanged, at 100.

This behavior is caused by the replication between Datastore servers. Replication is managed by Cloud Bigtable and Megastore, the underlying technologies for Datastore (see Additional Resources (#h.ywh7cedcuhkk) for more on details Bigtable and Megastore). The replication is executed with the Paxos (http://en.wikipedia.org/wiki/Paxos_(computer_science)) algorithm, which synchronously waits until a majority of the replicas have acknowledged the update request. The replica is updated with data from the request after a period of time. This time period is usually small, but there is no guarantee on its actual length. A query may read the stale data if it is executed before the update finishes.

In many cases, the update will have reached all the replicas very quickly. However, there are several factors that may, when compounded together, increase the time to achieve consistency. These factors include any datacenter-wide incidents that involve switching over a large number of servers between datacenters. Given the variation of these factors, it is impossible to provide any definitive time requirements for establishing full consistency.

The time required for a query to return the latest value is usually very short. However, in rare situations when the replication latency increases, the time can be much longer. Applications that use Datastore global queries should be carefully designed to handle these cases gracefully.

The eventual consistency on reading entity values can be avoided by using a keys-only query, an ancestor query, or lookup by key (the get() method). We will discuss these different types of queries in more depth below.

An index may not yet be updated when a global query is executed. This means that, even though you may able to read the latest property values of the entities, the “list of entities” included in the query result may be filtered based on old index values.

To understand the impact of eventual consistency on reading an index, imagine a scenario where a new entity, Player, is inserted into Datastore. The entity has a property, Score, which has an initial value of 300. Immediately after the insertion, you execute a keys-only query to fetch all entities with a Score value greater than 0. You would then expect the Player entity, just recently inserted, to appear in the query results. Perhaps unexpectedly, instead, you may find that the Player entity does not appear in the results. This situation can occur when the index table for the Score property is not updated with the newly inserted value at the time of the query execution.

Remember that all the queries in Datastore are executed against index tables (/appengine/docs/python/datastore/indexes#Python_Index_definition_and_structure), and yet the updates to the index tables are asynchronous (/appengine/articles/life_of_write). Every entity update is, essentially, made up of two phases. In the first phase, the commit phase, a write to the transaction log is performed. In the second phase, data is written and indexes are updated. If the commit phase succeeds, then the write phase is guaranteed to succeed, though it might not happen immediately. If you query an entity before the indexes are updated, you may end up viewing data that is not yet consistent.

As a result of this two phase process, there is a time delay before the latest updates to entities are visible in global queries. Just as with entity value eventual consistency, the time delay is typically small, but may be longer (even minutes or more in exceptional circumstances).

The same thing can happen after updates as well. For example, suppose you update an existing entity, Player, with a new Score property value of 0, and executed the same query immediately afterwards. You would expect the entity not to appear in the query results because the new Score value of 0 would exclude it. However, due to the same asynchronous index update behavior, it is still possible for the entity to be included in the result.

The eventual consistency on reading an index can be only be avoided by using an ancestor query or lookup by key method. A keys-only query can not avoid this behavior.

In Datastore, there are only two APIs that provide a strongly consistent view for reading entity values and indexes: (1) the lookup by key method and (2) the ancestor query. If application

logic requires strong consistency, then the developer should use one of these methods to read entities from Datastore.

Datastore is specifically designed to provide strong consistency on these APIs. When calling either one of them, Datastore will flush all pending updates on one of the replicas and index tables, then execute the lookup or ancestor query. Thus, the latest entity value, based on the updated index table, will always be returned with values based on the latest updates.

The lookup by key call, in contrast to queries, only returns one entity or a set of entities specified by a key or a set of keys. This means that an ancestor query is the only way in Datastore to satisfy strong consistency requirement together with a filtering requirement. However, ancestor queries do not work without specifying an entity group.

As discussed at the beginning of this document, one of the benefits of Datastore is that developers can find an optimal balance between strong consistency and eventual consistency. In Datastore, an entity group (/appengine/docs/python/datastore/structuring_for_strong_consistency) is a unit with strong consistency, transactionality, and locality. By utilizing entity groups, developers can define the scope of strong consistency among the entities in an application. In this way, the application can maintain consistency inside the entity group while, at the same time, achieving high scalability, availability, and performance as a complete system.

An entity group is a hierarchy formed by a root entity and its children or successors.[1] (#ftnt1) To create an entity group, a developer specifies an ancestor path, which is, essentially, a series of parent keys prefixing the child key. The concept of entity group is illustrated in Figure 3. In this case, the root entity with the key “ateam” has two children with the keys “ateam/098745” and “ateam/098746”.

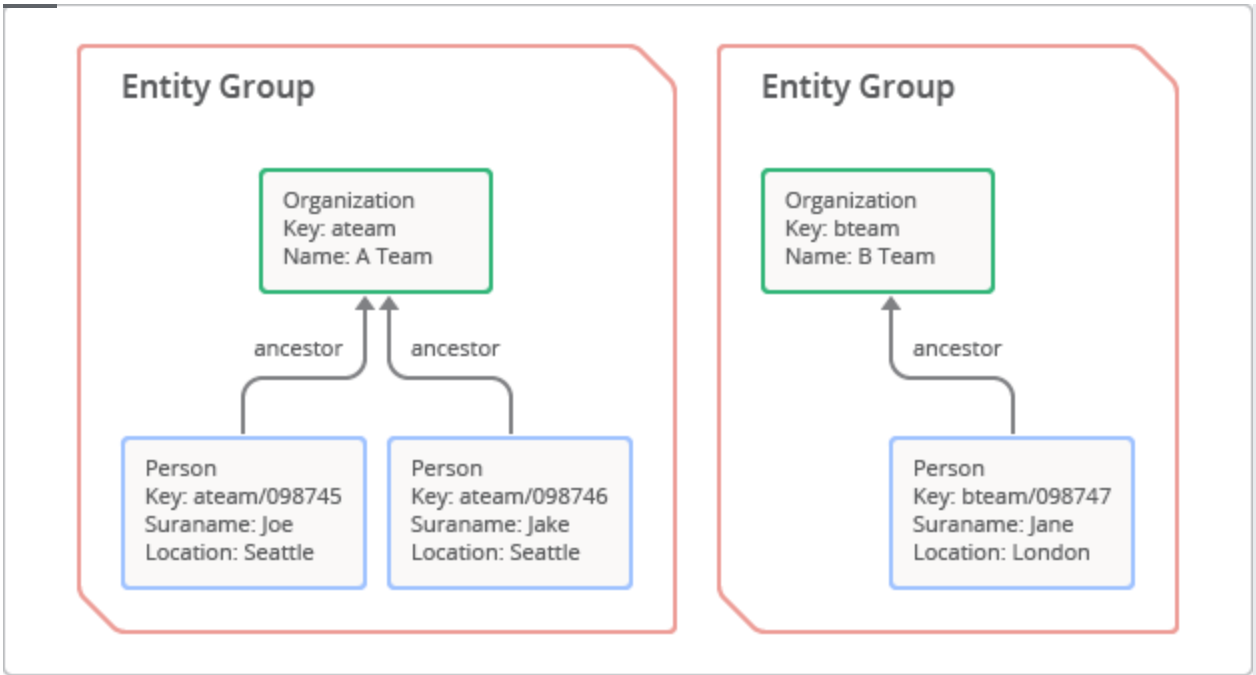

Figure 3: Schematic View of Entity Group Concept

Inside the entity group, the following characteristics are guaranteed:

- Strong Consistency
    - An ancestor query on the entity group will return a strongly consistent result. In this way, it reflects the latest entity values filtered by the latest index state.
- Transactionality
    - By demarcating a transaction programmatically, the entity group provides ACID (atomicity, consistency, isolation, and durability) characteristics in the transaction.
- Locality
    - Entities in an entity group will be stored at physically close places on Datastore servers, because all the entities are sorted and stored by the lexicographical order of the keys. This enables an ancestor query to rapidly scan the entity group with minimal I/O.

An ancestor query is a special form of query that only executes against a specified entity group. It executes with strong consistency. Behind the scenes, Datastore assures that all the pending replications and index updates are applied before executing the query.

This section describes how to use entity groups and ancestor queries in practice. In the following example, we consider the problem of managing data records for people. Suppose we have code that adds an entity of a specific kind followed immediately by a query on that kind. This concept is demonstrated by the example Python code below.

The problem with this code is that, in most cases, the query will not return the entity added in the statement above it. Since the query follows in the line following immediately after the insert, the index will not be updated when the query is executed. However, there is also a problem with validity of this use case: is there really a need to return a list of all people in one page with no context? What if there are a million people? The page would take too long to return.

The nature of the use case suggests that we should provide some context to narrow the query. In this example, the context that we will use will be the organization. If we do that, then we can use the organization as an entity group and execute an ancestor query, which solves our consistency problem. This is demonstrated with the Python code below.

This time, with the ancestor org specified in the GqlQuery, the query returns the entity just inserted. The example could be extended to drill down on an individual person by querying the person's name with the ancestor as part of the query. Alternatively, this could have also been done by saving the entity key and then using it to drill down with a lookup by key.

Entity groups can also be used as a unit for maintaining consistency between Memcache entries and Datastore entities. For example, consider a scenario where you count the number of Persons in each team and store them in Memcache. To make sure the cached data is consistent with the latest values in Datastore, you can use entity group metadata (/appengine/docs/python/datastore/metadataqueries#Python_Entity_group_metadata). The metadata returns the latest version number of specified entity group. You can compare the version number with the number stored in Memcache. Using this method you can detect a change in any of the entities in the entire entity group by reading from one set of metadata, instead of scanning all the individual entities in the group.

The approach of using entity groups and ancestor queries is not a silver bullet. There are two challenges in practice that make it hard to apply the technique in general, as listed below.

1. There is a limit of one update per second write for each entity group.
2. The entity group relationship can not be changed after entity creation.

An important challenge is that the system must be designed to contain the number of updates (or transactions) in each entity group. The supported limit is one update per second per entity group.[2] (#ftnt2) If the number of updates needs to exceed that limit then the entity group may be a performance bottleneck.

In the example above, each organization may need to update the record of any person in the organization. Consider a scenario where there are 1,000 people in the “ateam” and each person may have one update per second on any of the properties. As a result, there may be up to 1,000 updates per second in the entity group, a result which would not be achievable because of the update limit. This illustrates that it is important to choose an appropriate entity group design that considers performance requirements. This is one of the challenges of finding the optimal balance between eventual consistency and strong consistency.

A second challenge is the immutability of entity group relationships. The entity group relationship is formed statically based on key naming. It cannot be changed after creating the entity. The only available option for changing the relationship is to delete the entities in an entity group and recreate them again. This challenge prevents us from using entity groups to define ad-hoc scopes for consistency or transactionality dynamically. Instead, the consistency and transactionality scope are closely tied with the static entity group defined at design time.

For example, consider a scenario where you wish to implement a wire transfer between two bank accounts. This business scenario requires strong consistency and transactionality. However, the two accounts can not be grouped into one entity group last-minute or be based on a global parent. That entity group would create a bottleneck for the entire system that would hinder other wire transfer requests from being executed. So entity groups cannot be used in this way.

There is an alternative way to implement a wire transfer in a highly scalable and available way. Instead of placing all accounts in a single entity group, you can create an entity group for each account. By doing so, you can use transactions (/datastore/docs/concepts/transactions) to ensure ACID updates to both bank accounts. Transactions are a Datastore feature that allows you to create sets of operations with ACID characteristics for up to twenty-five entity groups. Note that within a transaction, you must use strongly consistent queries such as lookups by key and

ancestor queries. For more on the restrictions of transactions, see Transactions and entity groups (/datastore/docs/concepts/transactions#transactions_and_entity_groups).

If you already have an existing application with a large number of entities stored in Datastore, it may be difficult to incorporate entity groups afterwards in a refactoring exercise. It would require deleting all the entities and adding them within an entity group relationship. So, in data modeling for Datastore, it is important to make a decision on the entity group design in the early phase of the application design. Otherwise, you may be limited in refactoring to other alternatives to achieve a certain level of consistency, such as a keys-only query followed by a lookup-by-key, or by using Memcache.

A keys-only global query is a special type of global query that returns only keys without the property values of the entities. Since the return values are only keys, the query does not involve an entity value with a possible consistency problem. A combination of the keys-only, global query with a lookup method will read the latest entity values. But it should be noted that a keys-only global query can not exclude the possibility of an index not yet being consistent at the time of the query, which may result in an entity not being retrieved at all. The result of the query could potentially be generated based on filtering out old index values. In summary, a developer may use a keys-only global query followed by lookup by key only when an application requirement allows the index value not yet being consistent at the time of a query.

The Memcache service is volatile, but strongly consistent. So, by combining Memcache lookups and Datastore queries, it is possible to build a system that will minimize consistency issues most of the time.

For example, consider the scenario of a game application that maintains a list of Player entities, each with a score greater than zero.

- For insert or update requests, apply them to the list of Player entities in Memcache as well as Datastore.

- For query requests, read the list of Player entities from Memcache and execute a keys-only query on Datastore when the list is not present in Memcache.

The returned list will be consistent whenever the cached list is present in Memcache. If the entry has been evicted, or the Memcache service is not available temporarily, the system may need to read the value from a Datastore query that could possibly return an inconsistent result. This technique can be applied to any application that tolerates a small amount of inconsistency.

There are some best practices when using Memcache as a caching layer for Datastore:

- Catch Memcache exceptions and errors to maintain the consistency between the Memcache value and the Datastore value. If you receive an exception when updating the entry on Memcache, make sure to invalidate the old entry in Memcache. Otherwise there may be different values for an entity (an old value in Memcache and a new value in Datastore).
- Set an expiration period (/appengine/docs/python/memcache/#Python_How_cached_data_expires) on the Memcache entries. It is recommended to set short time periods for the expiration of each entry to minimize the possibility of inconsistency in the case of Memcache exceptions.
- Use the compare-and-set (/appengine/docs/python/memcache/#Python_Using_compare_and_set_in_Python) feature when updating the entries for concurrency control. This will help ensure that simultaneous updates on the same entry will not interfere with each other.

The suggestions made in the previous section only lessen the possibility of inconsistent behavior. It is best to design the application based on entity groups and ancestor queries when strong consistency is required. However, it may not be feasible to migrate an existing application, which may include changing an existing data model and application logic from global queries to ancestor queries. One way to achieve this is by having a gradual transition process, such as the following:

1. Identify and prioritize the functions in the application that require strong consistency.
2. Write new logic for insert() or update() functions using entity groups in addition to (rather than replacing) existing logic. In this way, any new inserts or updates on both new entity groups and old entities can be handled by an appropriate function.

3. Modify the existing logic for read or query functions ancestor queries are executed first if a new entity group exists for the request. Execute the old global query as fallback logic if the entity group does not exist.

This strategy allows for a gradual migration from an existing data model to a new data model based on entity groups that minimizes the risk of issues caused by eventual consistency. In practice, this approach is dependent on specific use cases and requirements for its application to an actual system.

At present, it is difficult to detect a situation programmatically when an application has deteriorated consistency. However, if you do happen to determine through other means that an application has deteriorated consistency, then it may be possible to implement a degraded mode that could be turned on or off to disable some areas of application logic that require strong consistency. For example, rather than showing an inconsistent query result on a billing report screen, a maintenance message for that particular screen could be shown instead. In this way, the other services in the application can continue serving, and in turn, reduce the impact to the user experience.

In a large application with millions of users or terabytes of Datastore entities, it is possible for inappropriate usage of Datastore to lead to deteriorated consistency. Such practices include:

- Sequential numbering in entity keys
- Too many indexes

These practices do not affect small applications. However, once the application grows very large, these practices increase the possibility of longer times needed for consistency. So it is best to avoid them at the early stages of application design.

Before the release of App Engine SDK 1.8.1, Datastore used a sequence of small integer IDs with generally consecutive patterns as the default auto-generated key names. In some

documents this is referred to as a “legacy policy” for creating any entities that have no application specified key name. This legacy policy generated entity key names with sequential numbering, such as 1000, 1001, 1002, for example. However, as we have discussed earlier, Datastore stores entities by the lexicographical order of the key names, so that those entities will be are very likely stored on the same Datastore servers. If an application attracts really large traffic, this sequential numbering could cause a concentration of operations on a specific server, which may result in longer latency for consistency.

In App Engine SDK 1.8.1, Datastore introduced a new ID numbering method with a default policy that uses scattered ID’s (see reference (/appengine/docs/python/datastore/entities#Python_Assigning_identifiers) documentation). This default policy generates a random sequence of ID’s up to 16 digits long that are approximately uniformly distributed. Using this policy, it is likely that the traffic of the large application will be better distributed among a set of Datastore servers with reduced time for consistency. The default policy is recommended unless your application specifically requires compatibility with the legacy policy.

If you do explicitly set key names on entities, the naming scheme should be designed to access the entities evenly over the whole key name space. In other words, do not concentrate access in a particular range as they are ordered by the lexicographical order of key names. Otherwise, the same issue as with the sequential numbering may arise.

To understand uneven distribution of access over the keyspace, consider an example where entities are created with the sequential key names as shown in the following code:

The application access pattern may create a “hot spot” over a certain range of the key names, such as having concentrated access on recently created Person entities. In this case, the frequently accessed keys will all have higher ID’s. The load may then be concentrated on a specific Datastore server.

Alternatively, to understand even distribution over the keyspace, consider using long random strings for key names. This is illustrated in the following example:

Now the recently created Person entities will be scattered over the keyspace and on multiple servers. This assumes that there is a sufficiently large number of Person entities.

In Datastore, one update on an entity will lead to update on all indexes defined for that entity kind (see Life of a Datastore Write (/appengine/articles/life_of_write) for details). If an application uses many custom indexes, one update could involve tens, hundreds, or even thousands of updates on index tables. In a large application, an excessive use of custom indexes could result in increased load on the server and may increase the latency to achieve consistency.

In most cases, custom indexes are added to support requirements such as customer support, troubleshooting, or data analysis tasks. BigQuery (/bigquery/) is a massively scalable query engine capable of executing ad-hoc queries on large datasets without pre-built indexes. It is better suited for use cases such as customer support, troubleshooting, or data analysis that require complex queries than Datastore.

One practice is to combine Datastore and BigQuery to fulfill different business requirements. Use Datastore for online transactional processing (OLTP) required for core application logic and use BigQuery for online analytical processing (OLAP) for backend operations. It may be necessary to implement a continuous data export flow from Datastore to BigQuery to move the data necessary for those queries.

Besides an alternate implementation for custom indexes, another recommendation is to specify unindexed properties explicitly (see Properties and value types (/datastore/docs/concepts/entities#properties_and_value_types)). By default, Datastore will create a different index table for each indexable property of an entity kind. If you have 100 properties on a kind, there will be 100 index tables for that kind, and an additional 100 updates on each update to an entity. A best practice, then, is to set properties unindexed where possible, if they are not needed for a query condition.

Besides reducing the possibility of having increases times for consistency, these index optimizations may result in quite a large reduction of Datastore storage costs (/datastore/docs/concepts/storage-size) in a large application which heavily uses indexes.

Eventual consistency is an essential element of non-relational databases that allows developers to find an optimal balance between scalability, performance, and consistency. It is important to understand how to handle the balance between eventual and strong consistency to design an optimal data model for your application. In Datastore, the use of entity groups and ancestor queries is the best way to guarantee strong consistency over a scope of entities. If your application cannot incorporate entity groups because of the limitations described earlier, you may consider other options such as using keys-only queries or Memcache. For large applications, apply best practices such as the use of scattered IDs and reduced indexing to decrease the time required for consistency. It may also be important to combine Datastore with BigQuery to fulfill business requirements for complex queries and to reduce the usage of Datastore indexes as far as possible.

The following resources provide more information about the topics discussed in this document:

- Google App Engine: Storing Data (/appengine/docs/python/datastore/)
- Datastore Overview (/datastore/docs/concepts/overview)
- Google Cloud Platform Blog (http://googlecloudplatform.blogspot.com/)
- Cloud SQL (/sql/)
- Using Python App Engine with Cloud SQL (/appengine/training/cloud-sql/)
- Bigtable: A Distributed Storage System for Structured Data (http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/bigtable-osdi06.pdf)
- App Engine 1.5.2 SDK Released (http://googleappengine.blogspot.com/2011/07/app-engine-152-sdk-released.html)

- Megastore: Providing Scalable, Highly Available Storage for Interactive Services (http://cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf)

[1] (#ftnt_ref1) An entity group can even be formed by specifying only one key of the root or parent entity, without storing the actual entities for the root or parent, because the entity group functions are all implemented based on relationships between keys.

[2] (#ftnt_ref2) The supported limit is one update per second per entity group outside transactions, or one transaction per second per entity group. If you aggregate multiple updates into one transaction, then you are limited to a maximum transaction size of 10 MB and the maximum write rate of Datastore server.