Google Cloud

Best Practices for Using Datastore

Let's say you want to store your applications data. And the data is structured or semi-structured, but not relational.

You want to get up and running quickly. But you want to make sure that the database option that you choose will be highly scalable and meets the needs of your application.

Datastore is the ideal solution for this use case. Datastore can scale from zero to millions of requests per second without you changing configuration or adding nodes to a cluster.

In this module, Best Practices for using Datastore, you'll learn key concepts about data objects, such as kinds, entities, keys, and attributes. You'll learn and apply best practices related to queries, built in and composite indexes, inserting and deleting data, transactions, and error handling.

A common question application developers have is, how do I bulk load data into Datastore? We will explore how to use Dataflow to bulk load data into Datastore.

Datastore = Firestore in Datastore mode

- Fully backward compatible with original Datastore but uses Firestore's improved storage layer.
- The Datastore page is used to manage the database.
- A project can have only a Firestore Native mode database or Datastore mode database, but not both.
- How to decide:
  - Choose Datastore mode when creating a new server application.
    - Automatically scales to millions of writes per second.
  - Choose Native mode for new mobile and web apps or when requiring real-time and offline features.
    - Automatically scales to millions of concurrent clients.

Firestore in Datastore mode uses Datastore system behavior but stores and reads data from Firestore's improved storage layer.

Firestore in Datastore mode is backward compatible with Datastore, but Firestore's native data model, real-time updates, and mobile and web client library features cannot be used with Datastore mode.

To access all of the new Firestore features, you must use Firestore in Native mode. When you create a new Firestore database, you must select a database mode, and you cannot use both Native mode and Datastore mode in the same project.

Datastore mode should typically be used for server applications. Native mode should be used for new mobile and web apps, or when requiring real-time and offline features.

Datastore concepts

- Data objects are called *entities.*

- Entities are made up of one or more *properties.*

- Each entity has a *key* that uniquely identifies it, composed of:
  - *Namespace*
  - Entity *kind*
  - *Identifier* (either a string or numeric ID)
  - *Ancestor Path* (optional)

- Operations on one more entities are called *transactions.*

Kind: Customer

Entity

Key

Property
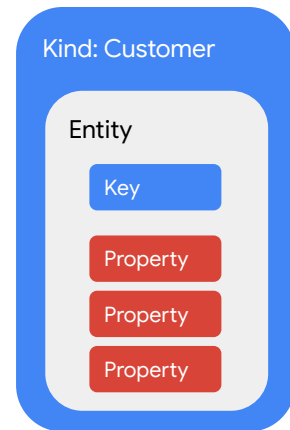
Property

Property

Data objects in Datastore are called entities and they are made up of one or more properties. Properties can have one or more values.
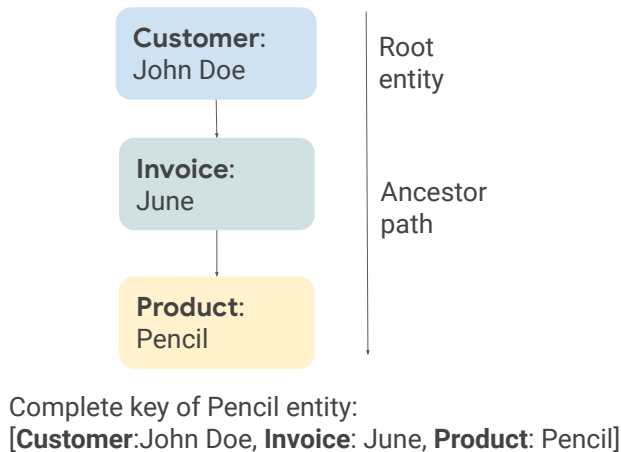
Each entity has a key that uniquely identifies it, composed of a namespace, the entity's kind, an identifier, and an optional Ancestor Path.

Operations on one or more entities are called transactions and are atomic.

[https://cloud.google.com/datastore/docs/concepts/entities]
[https://cloud.google.com/datastore/docs/concepts/transactions]

You can specify ancestors of an entity

Customer: John Doe — Root entity

Invoice: June — Ancestor path

Product: Pencil

Complete key of Pencil entity:
[**Customer**:John Doe, **Invoice**: June, **Product**: Pencil]

When you create an entity, you can specify another entity as its parent. An entity without a parent is a root entity.

An entity's parent, the parent's parent, and so on, are the entity's ancestors. An entity's child, the child's child, etc. are the entity's descendants.

The sequence of entities from the root entity to a specific entity forms the ancestor path.

The example shows the root entity John Doe with kind Customer. The complete key for the Pencil entity includes the kind-identifier pairs Customer John Doe, Invoice June, and Product Pencil.

Datastore has two types of indexes

Built-in indexes	Composite indexes
- Automatically pre-defined indexes for each property of each entity kind - Are suitable for simple types of queries	- Index multiple property values for indexed entity - Support complex queries - Are defined in an index configuration file

Datastore has two types of indexes: built-in and composite indexes.

By default, an ascending and descending index is predefined for each property of each entity kind. These are called built-in indexes. These single property indexes are sufficient to perform many simple queries, such as equality-only queries and simple inequality queries. These indexes are assumed, and do not appear in the indexes page of the Google Cloud Console.

For more complex queries, you must manually define composite indexes. Composite indexes are composed of multiple properties.

If a property will never be needed for a query, exclude the property from indexes. Unnecessarily indexing a property will result in increased latency to achieve consistency and increased storage costs for indexes.

You should avoid having too many composite indexes. Excessive use of composite indexes also results in increased latency and storage costs. If you need to execute ad hoc queries on large datasets without previously defined indexes, use BigQuery.

Do not index properties with monotonically increasing values (such as a NOW() timestamp). Maintaining such an index could lead to hotspots that impact Datastore latency for applications with high read and write rates.

[https://cloud.google.com/datastore/docs/concepts/indexes]

Create and delete your composite indexes

- Defined in a configuration file named `index.yaml`
- To create a composite index:
  - Add index definition to `index.yaml`
  - Run `gcloud datastore indexes create`
- To delete a composite index:
  - Remove indexes you no longer need from `index.yaml`
  - Run `gcloud datastore indexes cleanup`

Composite indexes are defined in the applications index configuration file (index.yaml). Composite indexes are viewable but not editable through the Cloud Console.

To deploy a composite index, modify the index.yaml configuration file to include all properties you want to index. Run "gcloud datastore indexes create" to create the new index. Depending on how much data is already in Datastore, creating the index may take a significant amount of time.

Queries that are run before the index has finished building will result in an exception.

When you change or remove an existing index from the index configuration, the original index is not deleted from Datastore automatically. When you're sure that old indexes are no longer needed, use "gcloud datastore indexes cleanup". That command deletes all indexes for the production Datastore instance that are not mentioned in the local version of index.yaml.

You can check the status of indexes in the Cloud Console.

[https://cloud.google.com/datastore/docs/tools/indexconfig]

Datastore as compared to relational databases

Datastore:
- Is designed to automatically scale to very large data sets.
- Doesn't support join operations, inequality filtering on multiple properties, or filtering on data based on results of a subquery.
- Doesn't require entities of the same kind to have a consistent property set.

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

The table compares concepts in Datastore with standard relational database concepts.

One key difference between Datastore and a relational database is that Datastore is designed to automatically scale to handle very large data sets, allowing applications to maintain high performance as they receive more traffic. Datastore writes scale by automatically distributing data as necessary. Datastore reads scale by only supporting queries whose performance scales with the size of the result set (as opposed to the data set). A query whose result set contains 100 entities performs the same whether it searches over a hundred entities or a million.

All queries are served by previously built indexes, so the types of queries that can be executed are more restrictive than those allowed on a relational database with SQL. Datastore does not include support for join operations, inequality filtering on multiple properties, or filtering on data based on the results of a subquery.

Unlike relational databases, datastore is schemaless. It doesn't require entities of the same kind to have a consistent property set. If you wish to enforce a specific set of properties for a particular kind, you must enforce that in your own application code.

[https://cloud.google.com/datastore/docs/concepts/overview#comparison_with_traditional_databases]

Demo

Explore Datastore

1. Create an App Engine application

2. Create Datastore entities

3. Query Datastore

4. Query Datastore using GQL

Demo 03a: Exploring Datastore

Design your app with these considerations in mind

- Use UTF-8 characters for:
  - Namespace names
  - Kind names
  - Property names
  - Key names

- Avoid forward slash (/) in:
  - Kind names
  - Custom key names

```
key = client.key('Task',
'sample_task')
```
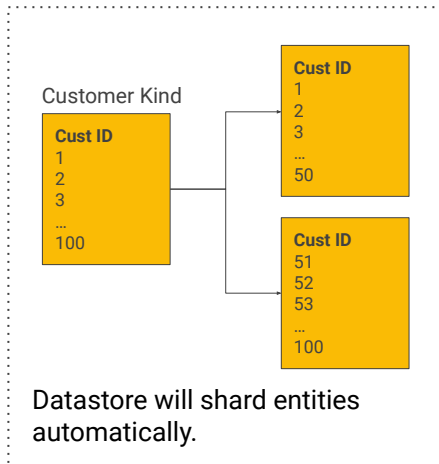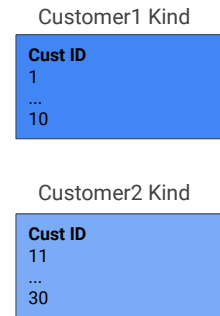✔️

```
key = client.key('Task',
'sample/task')
```
❌

Design your application using Cloud Datastore with these considerations in mind. Always use UTF-8 characters for namespace names, kind names, property names, and custom key names. Non-UTF-8 characters using these names can interfere with Cloud Datastore functionality.

Do not use a forward slash (/) in kind names or custom key names. Forward slashes in these names could interfere with future functionality.

Use sharding to increase rate of writes

Customer Kind

Cust ID
1
2
3
...
100

Cust ID
1
2
3
...
50

Cust ID
51
52
53
...
100

Datastore will shard entities automatically.

You can shard manually if the number of writes exceeds Datastore limits.

Customer1 Kind

Cust ID
1
...
10

Customer2 Kind

Cust ID
11
...
30

Sharding breaks up an entity into smaller pieces. Datastore will shard Entities automatically, distributing the data.

A single entity in Datastore should not be updated too rapidly. You should design your application so that it will not need to update an entity more than once per second, which is currently the maximum sustained write rate to a single entity. You can surpass the limit in short bursts, but sustained writes of over 1 per second to a single entity increases latency and may cause contention errors.

When you need to perform a high rate of sustained writes to an entity, though, you may choose to manually shard your entities into entities of different kinds, but using the same key.

[Datastore Best Practices: https://cloud.google.com/datastore/docs/best-practices
High Read/Write Rates to a Narrow Key Range:
https://cloud.google.com/datastore/docs/best-practices#high_readwrite_rates_to_a_narrow_key_range]

Shard counters to avoid contention with high writes

- Reduce contention by building a sharded counter, breaking the counter up into N different counters in N entities.

- To increment, pick a shard at random and increment its counter.

- To retrieve the count, read all of the sharded entities and sum their individual counts.

Counters are an example of a property that may need to be consistently updated more than one time per second.

To update an entity faster than the recommended one time a second, you can reduce contention by sharding the counter.
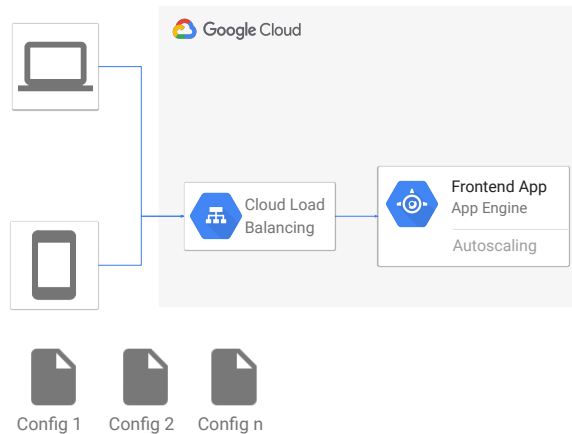
You can break the counter up into n different counters. When you want to increment the counter, you pick one of the shards at random and increment it. To know the total count, you read all of the counter shards and sum up the individual counts. Increasing the number of shards will increase the throughput you will have for increments on your counter.

[https://cloud.google.com/appengine/articles/sharding_counters]

Use replication to read a portion of the key range

Use replication to read a portion of the key range at a higher rate.

You can store N copies of the same entity, allowing an N times higher rate of reads.

Google Cloud

Cloud Load Balancing

Frontend App
App Engine

Autoscaling

Config 1   Config 2   Config n

If your application is bound by read performance, replication may be a better option for your application.

You can use replication if you need to read a portion of the key range at a higher rate than Datastore permits. Using this strategy, you would store N copies of the same entity, allowing an N times higher rate of reads than is supported by a single entity.

One standard use case is a static config file. In this case, your application could have a static number of config objects. You would create n copies of config, called config 1 through config n. When loading, the application would pick a random number r, between 1 and n, and it would retrieve that entity.

Things to remember when designing your application's operations

- Use batch operations for reads, writes, and deletes

- Roll back failed transactions

- Use asynchronous calls

Here are a few things to remember when designing operations for your application.

Use batch operations for your reads, writes, and deletes, instead of using single operations. Batch operations allow you to perform multiple operations on multiple objects with the same overhead as a single operation.

If a transaction fails, try to roll back the transaction. Having a rollback in place will minimize retry latency for concurrent requests of the same resources in a transaction. If an exception occurs during a rollback, it is not necessary to retry the rollback operation.

Where available, use asynchronous calls to minimize latency impact. For example, if you have an application that needs two or more separate lookups or queries before it can render a response, it is faster to make the requests asynchronously so they can be performed in parallel.

Use query types based on your needs

Keys-only	Projection	Ancestor	Entity
• Retrieve only the key • Return results at lower latency and cost (free)	• Retrieve specific properties from an entity • Retrieve only the properties included in the query filter • Return results at lower latency and cost (free)	• Limits results to the specified entity and its descendants	• Retrieve an entity kind, zero or more filters, and zero or more sort orders
`SELECT __key__ FROM Task`	`SELECT priority, percent_complete FROM Task`	`SELECT * FROM Task WHERE __key__ HAS ANCESTOR KEY(TaskList, 'default')`	`SELECT * FROM Task WHERE done = FALSE`

If you need to access only the key from query results, use a keys-only query. Likewise, if you need to access only specific properties from an entity or you need to access only the properties that are included in the query filter, use a projection query. Keys-only and projection queries return results at lower latency and cost than retrieving entire entities. In addition, keys-only and projection queries are classified as small operations, which are free if billing is enabled for your project. There is also a free quota for small operations for projects where billing is not enabled. [https://cloud.google.com/datastore/pricing#small_operations]

If your data is strongly hierarchical, and contains parent-child relationships, you can easily return all children of a given entity by using ancestor queries. Ancestor queries were the only type of queries with strong consistency in earlier versions of Datastore, but all queries are now strongly consistent with Firestore in Datastore mode.

You can retrieve an entity kind, zero or more filters, and zero or more sort orders with an entity query. An entity satisfies the filter if it has a property of the given name whose value compares to the value specified in the filter specified by the comparison operation. The example returns Task entities that are marked *not done*.

[https://cloud.google.com/datastore/docs/reference/gql_reference]

Improve your query latency by using cursors instead of offsets

Integer offsets	Query cursors
- Don't return skipped entities to your application - Still retrieve the entities internally - Cause your application to be billed for read operations	- Retrieve a query's results in convenient batches - Don't incur the overhead of a query offset

In general you should avoid using offsets in your queries, and instead use cursors.

Using an offset only avoids returning the skipped entities to your application, but these entities are still retrieved internally. The skipped entities affect the latency of the query, and your application is billed for the read operations required to retrieve them.

Cursors still retrieve the queries in batches, but do not incur the latency and cost seen with offsets.

[For more information on query cursors, see: https://cloud.google.com/appengine/docs/standard/python/datastore/query-cursors]

Numeric IDs as keys

For keys that use numeric IDs:

- Do not use a negative number.

- Do not use the value 0.

- If you wish to manually assign numeric IDs to your entities, get a block of IDs using the allocateIds() method.

- Avoid monotonically increasing values.

For a key that uses a numeric ID, do not use a negative number for the ID. Negative numbers can interfere with sorting.

Using the value 0 for the ID will cause an automatically allocated ID to be created.

To assign your own numeric IDs manually to the entities you create, have your application obtain a block of IDs with the allocateIds() method. This will prevent Datastore from assigning one of your manual numeric IDs to another entity.

If you assign your own manual numeric ID or custom name to the entities you create, do not use monotonically increasing values such as 1, 2, 3, or Product1, Product2, Product3. Sequential numbering can lead to hotspots that impact Datastore latency. The allocateIds() method generates well-distributed sequences of numeric IDs, so use it to avoid hotspots.

Transaction design considerations

- Atomic:
  - All are applied or
  - None are applied

- Max duration: 270 sec

- Idle expiration: 60 sec

Can fail when:

- Too many concurrent modifications are attempted on the same entity.

- A resource limit is exceeded.

- Datastore encounters an internal error.

> Make your Datastore transaction idempotent whenever possible!

Transactions are a set of Datastore operations on one or more entities. They are guaranteed to be atomic, which means that either all operations in the transaction are applied or none of them are applied.

The maximum transaction time is 270 seconds, but Datastore will terminate a transaction if there is no activity for 60 seconds.

A transaction may fail when too many concurrent modifications are attempted on the same entity, the transaction exceeds a resource limit, or the Datastore database encounters an internal error.
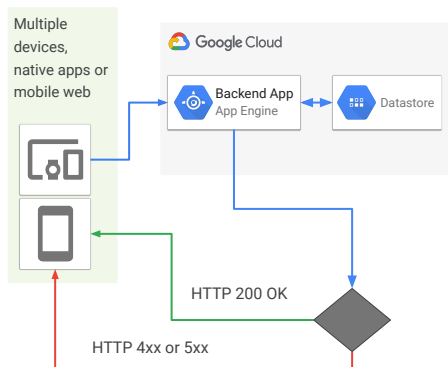
If your application receives an exception when committing a transaction, it does not always mean that the transaction failed. You can receive errors in cases where transactions have been committed and eventually will be applied successfully. Whenever possible, make your Datastore transactions idempotent so that if you repeat a transaction, the end result will be the same. An idempotent operation is one that has no additional effect if it is called more than once with the same input parameters.

[Refer to the documentation for more information on idempotence and Datastore transactions.
https://en.wikipedia.org/wiki/Idempotence
https://cloud.google.com/datastore/docs/concepts/transactions]

Design your application to handle errors

Multiple devices, native apps or mobile web

Google Cloud

Backend App
App Engine

Datastore

HTTP 200 OK

HTTP 4xx or 5xx

Error	Recommended Action
• ALREADY_EXISTS • FAILED_PRECONDITION • INVALID_ARGUMENT • NOT_FOUND • PERMISSION_DENIED • UNAUTHENTICATED	Do not retry without fixing the problem.
• RESOURCE_EXHAUSTED	Fix quota if exceeded. If quota was not exceeded, retry using exponential backoff.
• DEADLINE_EXCEEDED • UNAVAILABLE	Retry using exponential backoff.
• INTERNAL	Do not retry this request more than once.
• ABORTED	For a non-transactional commit: • Retry the request or structure your entities to reduce contention. For requests that are part of a transactional commit: • Retry the entire transaction or structure your entities to reduce contention.

When a Datastore request succeeds, the API will return a 200 OK status code, with the requested data in the body of the response.

Failures return a 4xx or 5xx status code with more specific information about the errors that caused the failure.
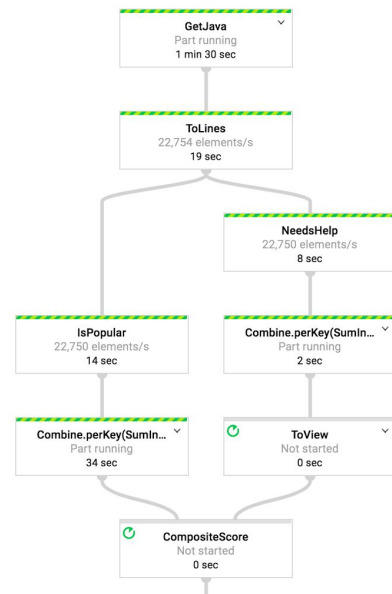
Errors should be classified by inspecting the value of the canonical error code. The table displays recommended actions per error code.

[For more information, see: https://cloud.google.com/datastore/docs/concepts/errors]

Demo 3b: Bulk Loading Datastore Data from Cloud Storage

Google created Dataflow as a way of unifying batch and stream processing. Google donated it to Apache where it's called Beam (batch and stream). In this demo, we will run the pipeline by using the local Dataflow Runner.

The image shows the demo pipeline in Dataflow.

For more information, see:
Apache Beam Documentation: https://beam.apache.org/documentation/
Programming Model for Cloud Dataflow SDK 2.x:
https://cloud.google.com/dataflow/model/programming-model-beam
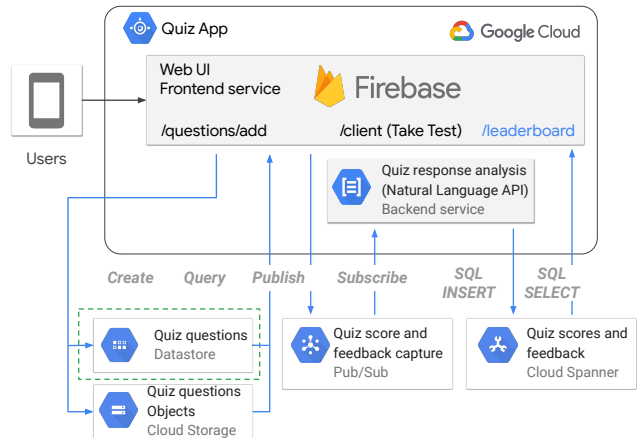
Lab

Storing Application Data in Datastore

Duration: 45 minutes

Lab objectives

Use Cloud Shell as your development environment

Integrate Datastore into the quiz application

To summarize, Datastore is a fully managed, no SQL database service that you can use to store structured or semi-structured application data. Data objects in Datastore are called entities. Each entity is of a particular kind. You can specify ancestor path relationships between entities to create entity groups.

Ancestor queries of entity groups give you a strongly consistent view of the data.

By creating entity groups, you can ensure that all related entities can be updated in a single transaction. Datastore automatically builds indexes for individual properties in an entity. To enable more complex queries with filters on multiple properties, you can create composite indexes. Datastore can scale seamlessly with zero down time, make sure to ramp up traffic gradually.

A general guideline is to use the 500-50-5 rule to ramp up traffic. You want to start with a base write rate of 500 writes per second and increase it by 50% every 5 minutes. Distribute your writes across a key range.

By applying the best practices that you have learned in this module, you can be confident that your web or mobile app is set up for success.