Best Practices for Using Cloud Datastore

Developing Applications with Google Cloud Platform

CLOUD DATASTORE, CLOUD BIGTABLE

QWIKLABS STORING APPLICATION DATA IN CLOUD DATASTORE

Google Cloud

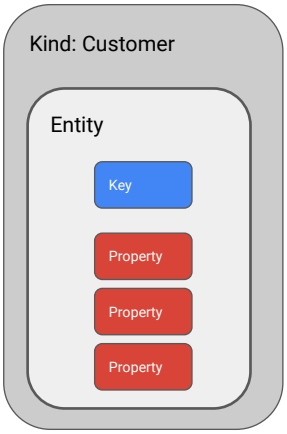
Version 2.0
Last modified: 2017-09-25

This presentation discusses considerations for building an application with Google Cloud Datastore.

For more information on Google Cloud Datastore, see:
https://cloud.google.com/datastore/docs/datastore-api-tutorial
For more information on Google Cloud Datastore Best Practices, see:
https://cloud.google.com/datastore/docs/best-practices

Cloud Datastore concepts

- Data objects are called *entities*.
- Entities 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:
  - *Namespace*
  - Entity *kind*
  - *Identifier* (either a string or numeric ID)
  - *Ancestor ID* (optional)
- Operations on one or more entities are called *transactions*.

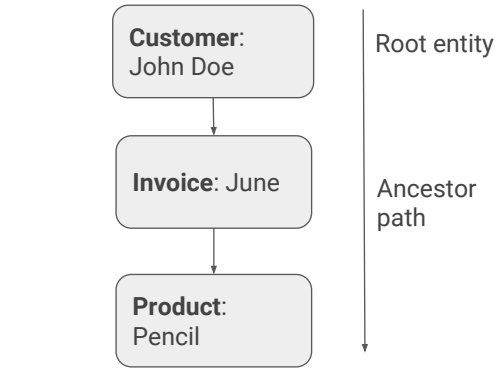


Google Cloud

Data objects in Cloud 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 kind, an identifier, and an optional ancestor ID. Operations on one or more entities are called transactions and are atomic.

For more information, see:
Entities: https://cloud.google.com/datastore/docs/concepts/entities
Transactions: https://cloud.google.com/datastore/docs/concepts/transactions

You can specify ancestors of an entity.



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

Google Cloud

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, etc. are its ancestors. An entity's child, the child's child, etc. are descendants.

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

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

For more information about Entities, properties, and keys see https://cloud.google.com/datastore/docs/concepts/entities

Datastore has two types of indexes

Built-in indexes	Composite indexes
• Automatically pre-define an index 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

Google Cloud

Built-in indexes are sufficient to perform many simple queries, such as equality-only queries and simple inequality queries. For more complex queries, an application must define composite, or manual, indexes.

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

Avoid having too many composite indexes. Excessive use of composite indexes could result in increased latency to achieve consistency and increased storage costs of index entries. If you need to execute ad hoc queries on large datasets without previously defined indexes, use Google BigQuery.

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

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

Create and delete your composite indexes

- To deploy a composite index:
  - Modify the `index.yaml` configuration file to include all properties to be indexed.
  - Run `gcloud datastore create-indexes.`
- To delete a composite index:
  - Modify the `index.yaml` configuration file to remove the indexes you no longer need.
  - Run `gcloud datastore cleanup-indexes.`

Google Cloud

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

To deploy a composite index, modify the index.yaml configuration file to include all properties you want to index. Run gcloud datastore create-indexes to create the new index. Depending on how much data is already in Cloud Datastore, creating the index could take some time. Queries that are run before the index has finished building will result in an exception.

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

For more information, see:
Index Configuration: https://cloud.google.com/datastore/docs/tools/indexconfig

Cloud Datastore concepts compared to relational databases

Cloud Datastore:

- Is designed to automatically scale to very large data sets by
  - Writing scale by distributing data
  - Reading scale
- Does not 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	Cloud 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

Google Cloud

The table shows concepts in Cloud Datastore that you would know from a relational database.

Key differences between Cloud Datastore and a relational database are that Datastore is designed to automatically scale to very large data sets, allowing applications to maintain high performance as they receive more traffic. Datastore maintains high performance by writing scale by automatically distributing data as necessary. Datastore also reads scale because the only queries supported are those whose performance scales with the size of the result set (as opposed to the data set). So a query whose result set contains 100 entities performs the same whether it searches over a hundred entities or a million. For this reason, some types of queries are not supported.

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. Cloud Datastore does not include support for join operations, inequality filtering on multiple properties, or filtering on data based on results of a subquery.

Cloud Datastore doesn't require entities of the same kind to have a consistent property set (although you can enforce such a requirement in your own application code).

For more information, see:
https://cloud.google.com/datastore/docs/concepts/overview#comparison_with_traditio

nal_databases

Demo: Explore Cloud Datastore

1. Create an App Engine application.
2. Create Cloud Datastore Entities.
3. Query Cloud Datastore.
4. Query Cloud Datastore using GQL.

Google Cloud

Demo 03a: Exploring Cloud Datastore

Design your application 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
- Avoid storing sensitive information in a Cloud Project ID

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



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


Google Cloud

Always use UTF-8 characters for namespace names, kind names, property names, and custom key names. Non-UTF-8 characters used in 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.

Avoid storing sensitive information in a Cloud Project ID. A Cloud Project ID might be retained beyond the life of your project.

The examples show creating a key with kind *Task* using a key name "sample_task" as the identifier (recommended) and “sample/task” (not recommended).

Design your application for scale

- The maximum write rate to an entity group is 1/second.
- Avoid high read or write rates to keys that are lexicographically close.
- Gradually ramp up traffic to new Cloud Datastore kinds or portions of the keyspace.
- Avoid deleting large numbers of Cloud Datastore entities across a small range of keys.
- For hot Cloud Datastore keys:
  - Use sharding to write to a portion of the key range at a higher rate.
  - Use replication to read a portion of the key range at a higher rate.

Google Cloud

By design, Datastore does not handle high amounts of writes to a single Entity Group. To design for this characteristic, the recommendation is to shard Entities. If you update an entity group too rapidly, then your Cloud Datastore writes will have higher latency, timeouts, and other types of errors—known as contention.

Avoid high read or write rates to Cloud Datastore keys that are lexicographically close. Because Cloud Datastore is built on top of Google's NoSQL database (Bigtable), it is subject to Bigtable's performance characteristics. Bigtable scales by sharding rows onto separate tables, and these rows are lexicographically ordered by key.

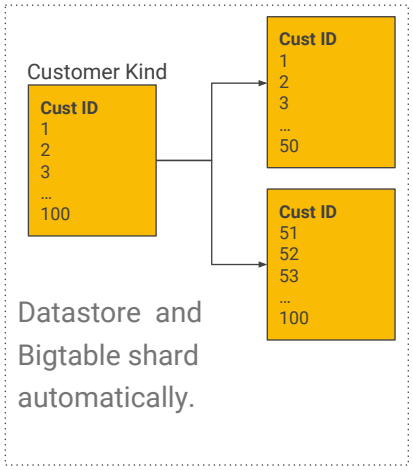
Gradually ramp up traffic to new Cloud Datastore kinds or portions of the keyspace. You should ramp up traffic to new Cloud Datastore kinds gradually in order to give Bigtable sufficient time to split tablets as the traffic grows.

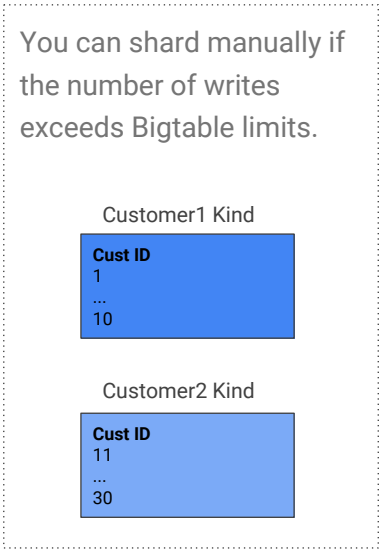
Avoid deleting large numbers of Cloud Datastore entities across a small range of keys. Bigtable periodically rewrites its tables to remove deleted entries and to reorganize your data so that reads and writes are more efficient. This process is known as a *compaction*. When performing a large number of deletes for a range of entities, if those entities have an index on a timestamp field, Datastore will store those entities closely together. When those entities are all deleted, performance for queries in that part of the index will be slower until compaction has been completed.

Use sharding or replication for hot Cloud Datastore keys. Replication can be used to

read a portion of the key range at a higher rate than Bigtable permits. Use sharding to write to a portion of the key range at a higher rate than Bigtable permits.

Use sharding to write to a portion of the key range





Google Cloud

Sharding splits a single entity into many. Datastore and Bigtable will shard automatically. If your application requires more writes than Bigtable's limits, you can shard manually.

Use sharding to write to a portion of the key range at a higher rate than Bigtable permits. Sharding breaks up an entity into smaller pieces.

In the example, because of the need to write frequently to a key range in the Customer kind, it has been sharded to improve performance. The sharding could theoretically be based on customer region or regions.

For more information, see:
Cloud 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

When sharding, remember:

- Transaction throughput is limited to 1 write/sec per entity group.
- Split frequently updated entities across multiple kinds.

Google Cloud

Keep the following in mind when sharding:

- Transaction throughput is limited to 1 write/second per entity group.
- Split frequently updated entities across multiple kinds.

For more information, see:
https://cloud.google.com/datastore/docs/best-practices#sharding_and_replication

Shard counters to avoid contention with high writes

Reduce contention by:

- Building a sharded counter (break the counter up into N different counters):
  - Pick a shard at random to increment the counter
  - To know the total count, read all counter shards and sum their individual counts

Increasing the number of shards will increase the throughput you will have for increments on your counter!

Google Cloud

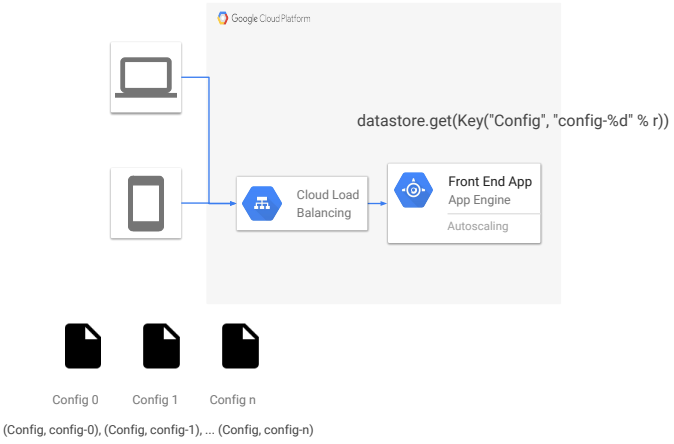
To update an entity faster than the recommended five times a second, you can shard counters. To reduce contention (serialized writes stack up and start to time out), you would build a sharded counter. You would break the counter up into n different counters. When you want to increment the counter, you pick one of the shards at random and increment it. To know the total count, read all of the counter shards and sum up individual counts. Increasing the number of shards will increase the throughput you will have for increments on your counter.

For more information, see:
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

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 Bigtable 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 (shown in the diagram) is a static config file, which may get loaded for each request. In this case, your application would have a static number of config objects (kind Config, named config-<number>), e.g.: (Config, config-0), (Config, config-1), ... (Config, config-n). When loading, the application would pick a random number r between 0–n and calls datastore.get(Key("Config", "config-%d" % r)).

When designing your application's operations, remember to:

- Use batch operations for:
  - Reads
  - Writes
  - Deletes
- Roll back failed transactions.
- Use asynchronous calls.

Google Cloud

Use batch operations for your reads, writes, and deletes instead of using single operations. Batch operations allow you to perform multiple operations 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 the result of a lookup() and the results of a query before it can render a response, the lookup() and the query do not have a data dependency, so there is no need to synchronously wait until the lookup() is completed before initiating the query.

Use query types based on your needs

Keys-only	Projection	Ancestor	Entity
• Retrieve only the key • Return results at lower latency and cost	• Retrieve specific properties from an entity • Retrieve only the properties included in the query filter • Return results at lower latency and cost	• Return strongly consistent results • Requires your data to be structured for strong consistency	• 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`

Google Cloud

If you need to access only the key from query results, use a keys-only query. A keys-only query returns results at lower latency and cost than retrieving entire entities.

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. A projection query returns results at lower latency and cost than retrieving entire entities.

If you need strong consistency for your queries, use an ancestor query. Note that ancestor queries require that you structure your data for strong consistency. An ancestor query returns strongly consistent results.

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.*

Examples of each type of query are provided in GQL. For more information on GQL, see: 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

Google Cloud

Do not use offsets; 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.

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

Design your keys with these considerations in mind

For keys that use numeric IDs:

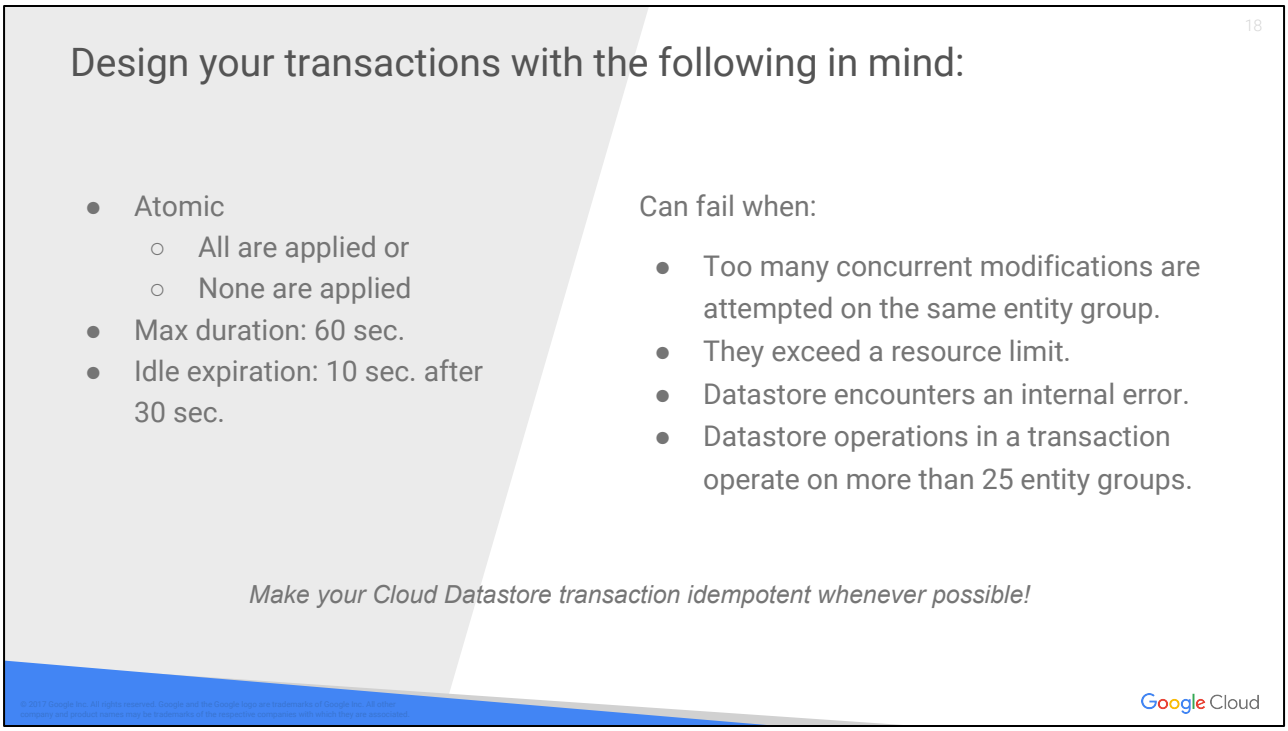
- Do not use a negative number.
- Do not use the value 0.
- Get a block of IDs using the allocateIds() method if you want to assign your own numeric IDs.
- Avoid monotonically increasing values.

For a key that uses a numeric ID, do not use a negative number for the ID and do not use the value 0 for the ID.

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 Cloud 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; this creates a Bigtable hotspot, where writes to new entities are always occurring at the end of the row key space, thereby preventing Bigtable from effectively distributing writes.

If an application generates large traffic, sequential numbering could lead to hotspots that impact Cloud Datastore latency. To avoid the issue of sequential numeric IDs, obtain numeric IDs from the allocateIds() method. The allocateIds() method generates well-distributed sequences of numeric IDs.



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.

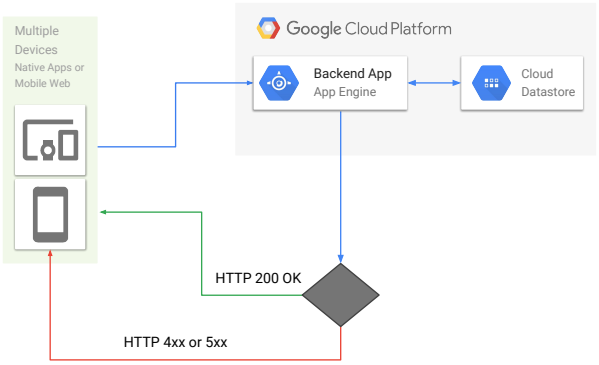
Maximum transaction time is 60 seconds, but if a transaction lasts more than 30 seconds, Datastore will terminate it if there is no activity for 10 seconds.

Transactions may fail when too many concurrent modifications are attempted on the same entity group, transactions exceed a resource limit, datastore encounters an internal error, or datastore operations in a transaction operate on more than 25 entity groups.

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 Cloud 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.

For more information, see:
Idempotence: https://en.wikipedia.org/wiki/Idempotence
Datastore transactions:
https://cloud.google.com/datastore/docs/concepts/transactions

Design your application to handle errors



Error	Recommended Action
ALREADY_EXISTS, FAILED_PRECONDITION, INVALID_ARGUMENT, NOT_FOUND, PERMISSION_DENIED, RESOURCE_EXHAUSTED, UNAUTHENTICATED	Do not retry without fixing the problem.
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.

Google Cloud

When a Google Cloud Datastore request succeeds, the API will return an HTTP 200 OK and the requested data in the body of the response.

Failures return an HTTP 4xx or 5xx 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.

Refer to “Design your application for scale” earlier in this module for approaches you can use to avoid scaling/contention errors.

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

You will build components of the online Quiz application in each lab