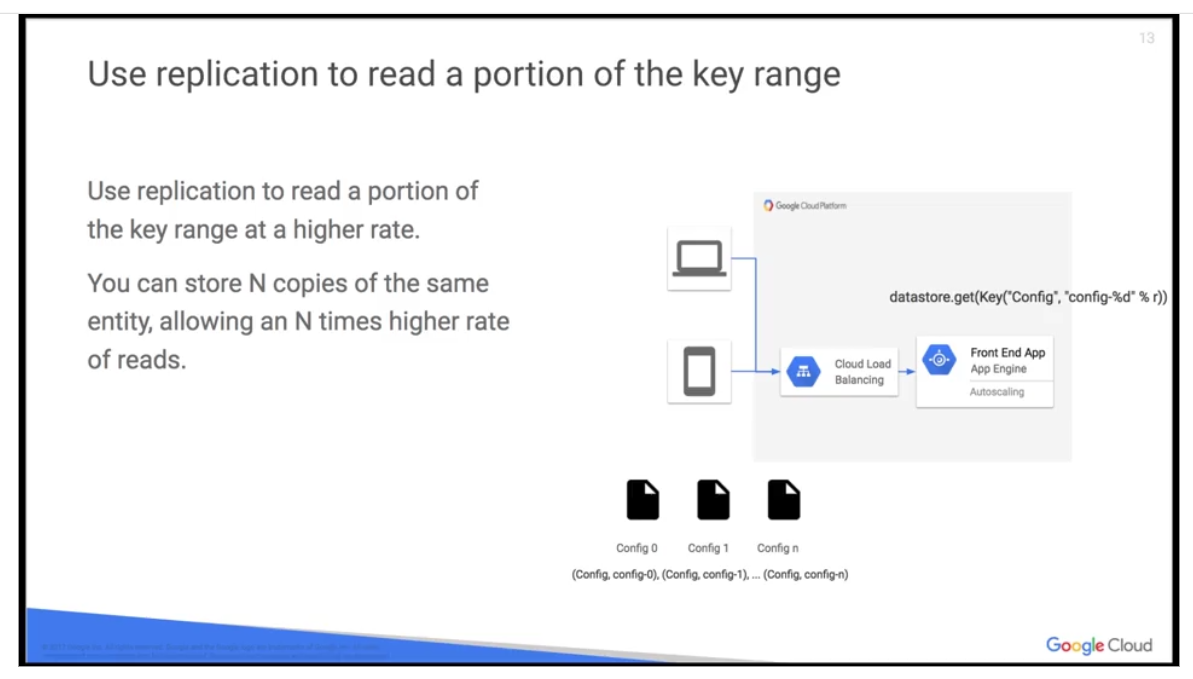
# Replication

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 rage 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.

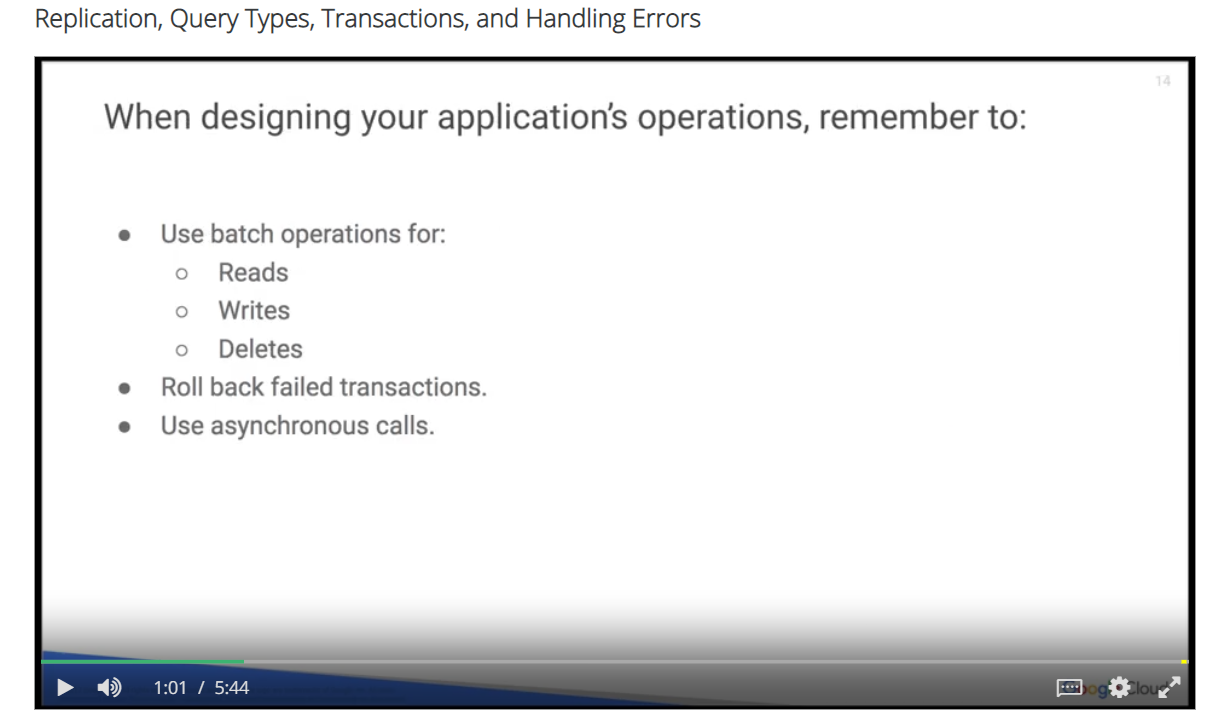


One standard use case, shown in the diagram, is a static config file, which would get loaded per request. In this case, your application would have a static number of config objects type kind Config, named config-number,

as seen here. So you would have (Config, config-0), (Config, config-1), all the way to (Config,

config-n). When loading, the application would pick a random number r between 0 and n, and it would call datastore.get(Key("Config",”config-%d” %r)) and then do some operations based on the number.

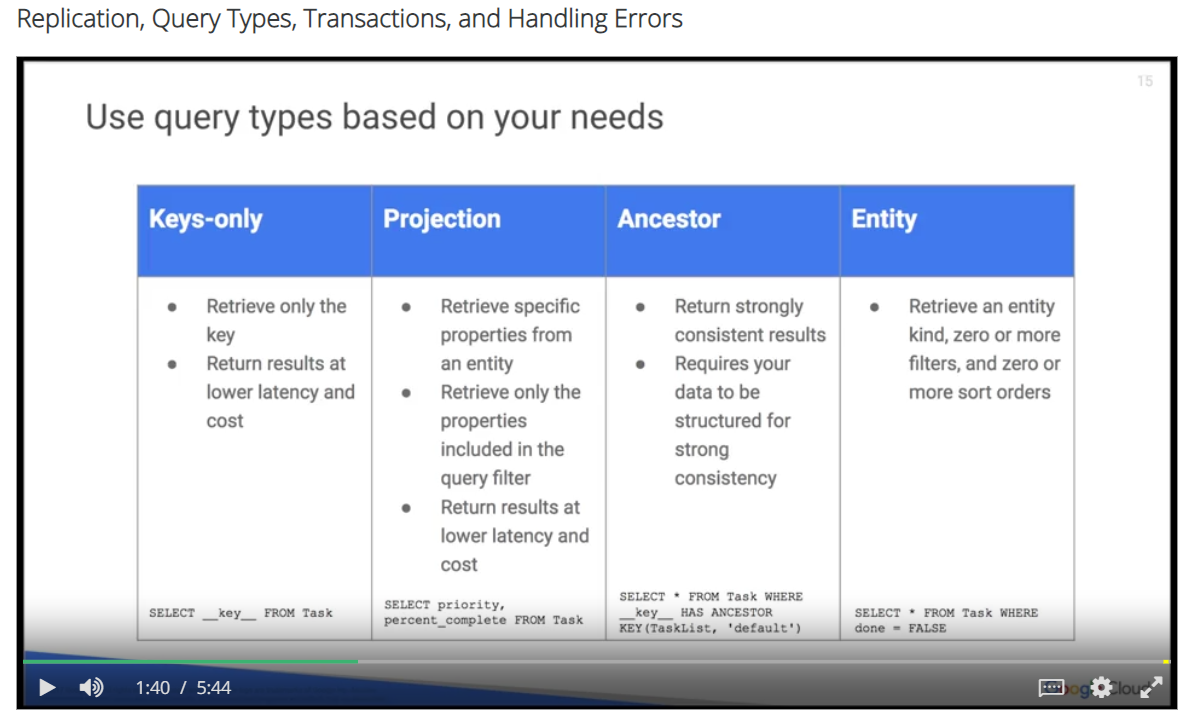
Batch



You should 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**.

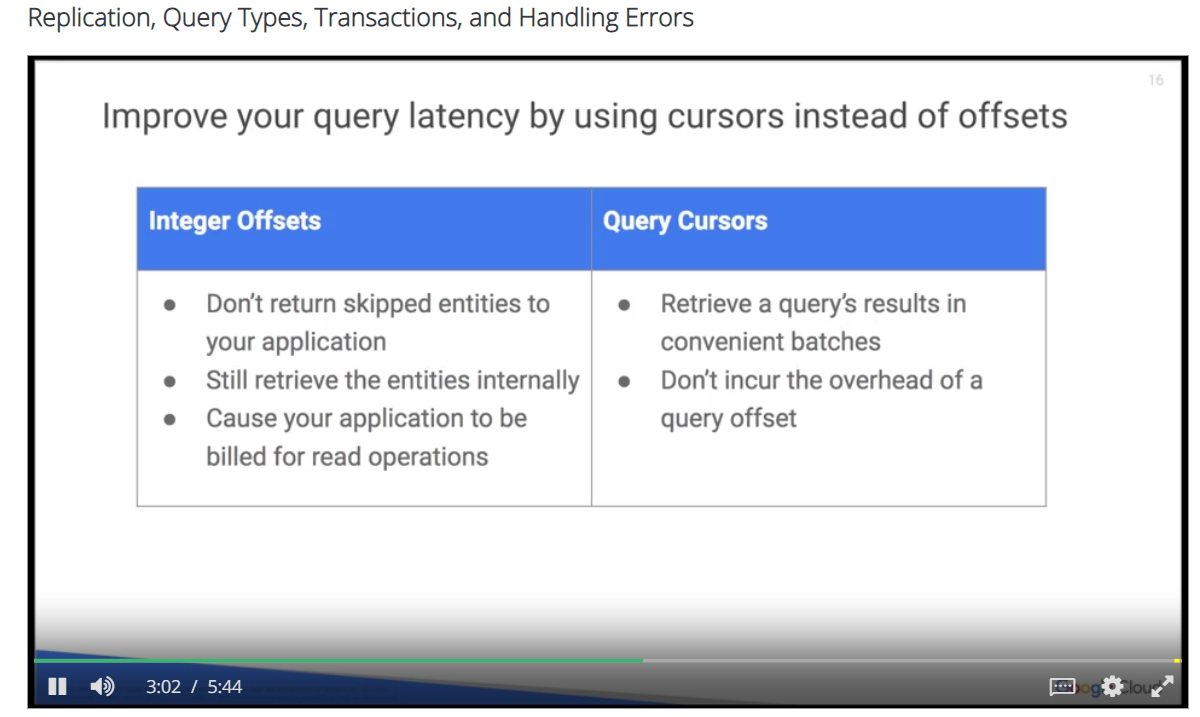
Having a roll back in place will minimize retry latency for concurrent requests of the same resources in a transaction. If an exception occurs during a roll back, it is not necessary to retry the rollback operation. Where available, use asynchronous calls to minimize latency impact.

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.

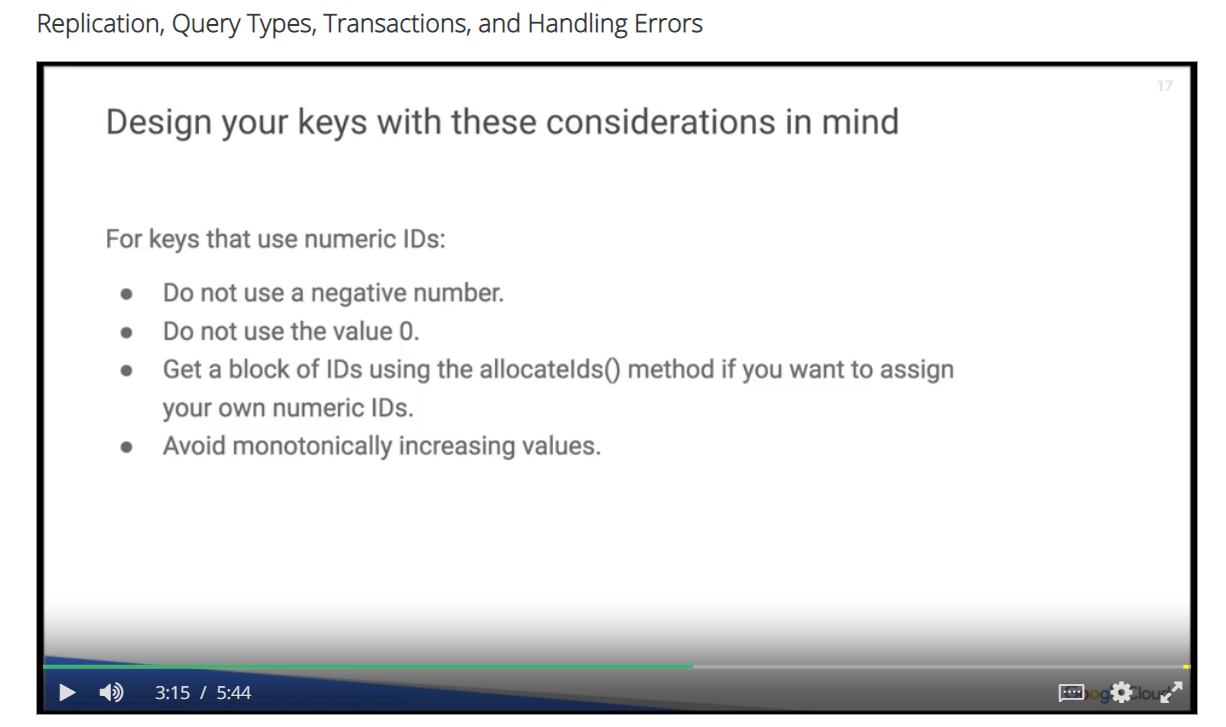


Use different query types based on your needs.

* If you need to access only the key from query results, use a keys-only query. A keys-only query returns a result at a lower latency and cost than retrieving entire entities. In the chart you will see an example of a key-only query.
* 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 a lower latency and cost that retrieving entire ententes. An example is shown of a projection query in the table.
* 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 return strongly consistent results. An example shown in the table as well.
* 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. More examples are available in the reference link provided in the downloads section below.

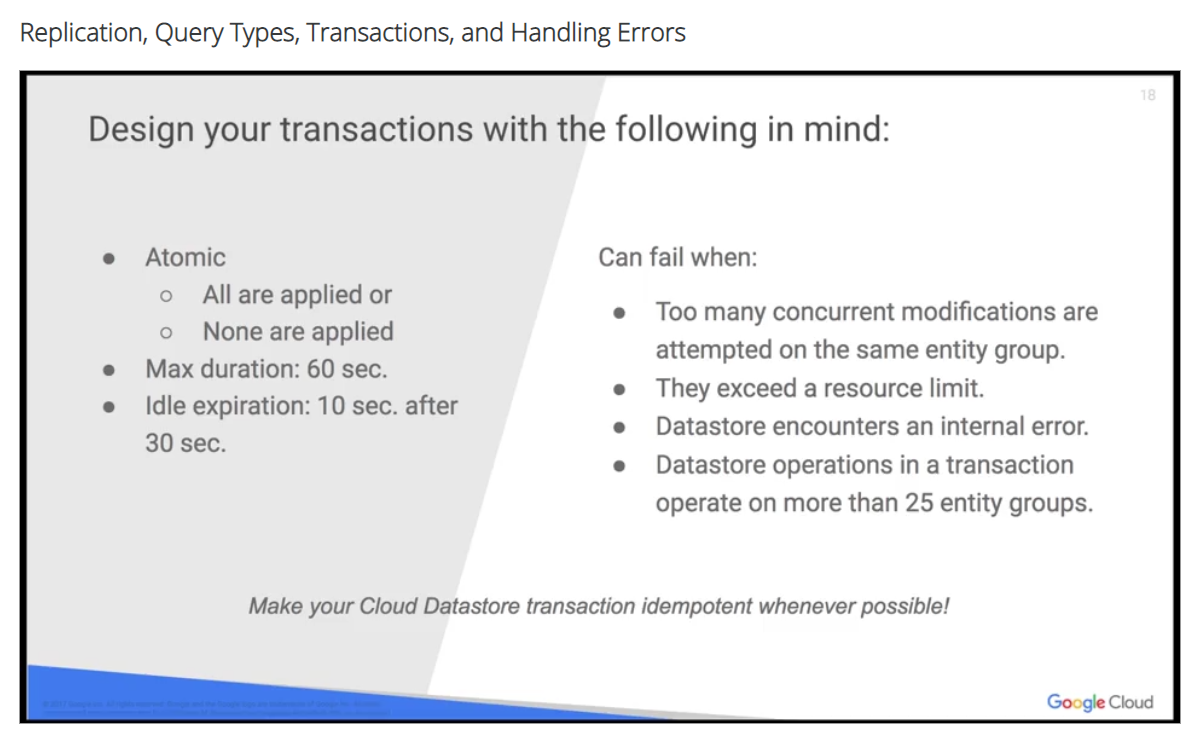


Do not use offsets, instead use cursors. Using an offset only avoids returning the skipped entities to your application, but these entities are still retrieved internally. The skipped entities affect the latency of the query, and your application is billed for the read operations required to retrieve them.

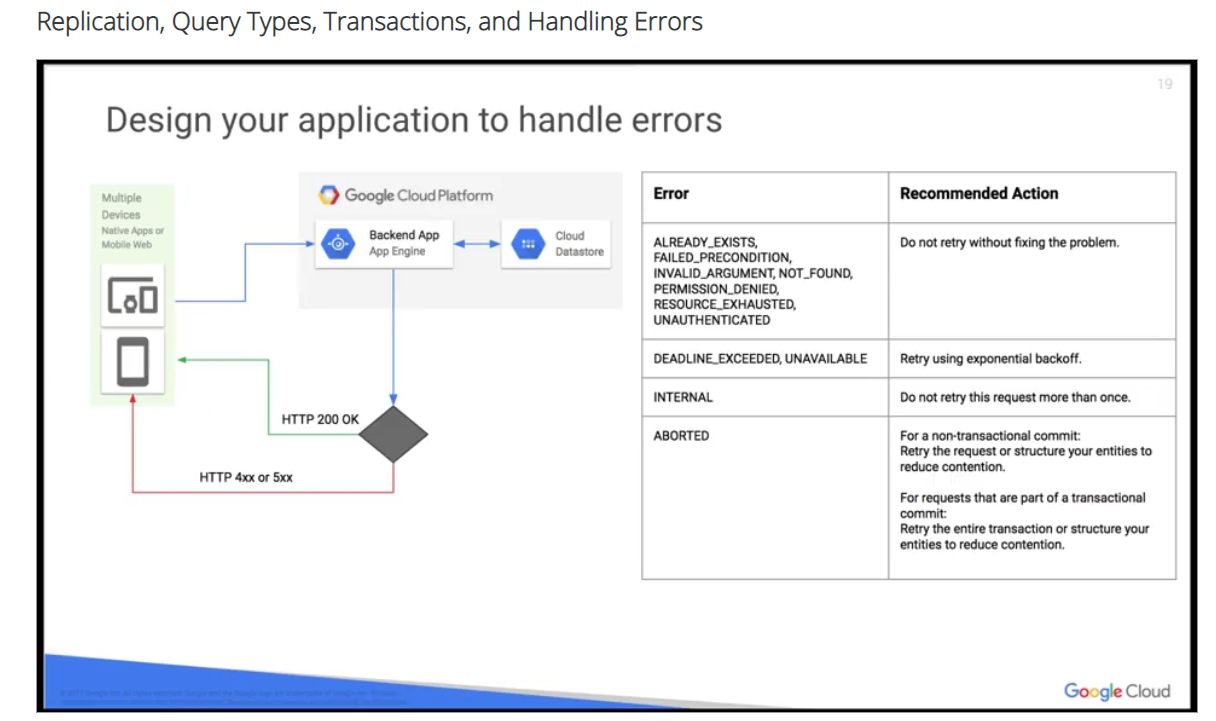


For a key that uses the numeric ID, do not use a negative number for the ID, and do not use the value 0 for the ID. To assign your own numeric IDs manually to the entities you create, have your application obtain a block

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 hot spot, where writes to new entities are always occurring at the end of the row key space, thereby preventing Bigtable from effectively distributing rights. 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 mentioned earlier. 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 obligation 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. 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 400, or 500, 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 the module, for approaches you can use to avoid scaling and contention errors.