Let's say you want to store your application's 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. Cloud Datastore is the ideal solution for this use case. Cloud Datastore can scale from zero to millions of requests per second without you changing configuration or adding nodes to a cluster.

In the module best practices for using Cloud Datastore you'll learn key concepts about data objects, such as **kinds, entities, keys, and attributes**. You will 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 Cloud Datastore? We will explore how to use **Cloud Dataflow to bulk load data into Cloud Datastore**.

# Comparison with traditional databases

<https://cloud.google.com/datastore/docs/concepts/overview>

While the Cloud Datastore interface has many of the same features as traditional databases, as a NoSQL database it differs from them in the way it describes relationships between data objects. Here’s a high-level comparison of Cloud Datastore and relational database concepts:

| Concept | Cloud Datastore | Cloud Firestore | Relational database |
| --- | --- | --- | --- |
|  |  |  |  |
| Category of object | Kind | Collection group | Table |
| One object | Entity | Document | Row |
| Individual data for an object | Property | Field | Column |
| Unique ID for an object | Key | Document ID | Primary key |

Cloud Datastore entities of the same kind can have different properties, and different entities can have properties with the same name but different value types.

# Best Practices for Using Cloud Datastore

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. Throughout this module please feel free to click any of the documentation links provided in the download pane below. 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 parents, 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 kind identify pairs, customer John Doe, invoice June, and the Pencil entity itself.

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 increase storage cost of index entries.

Avoid having too many composite indexes. Excessive use of composite indexes could result in increase latency to achieve consistency, and increase storage costs of index entries.

If you need to execute ad-hoc queries on large datasets without previously defined indexes you should use Google BigQuery. Do not index properties with monotonically increasing values, such as a now, or a date time stamp. Maintaining such an index could lead to hot spots that impact Cloud Datastore latency for applications with high read and write rates.

Composite indexes are defined in the application 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 the command datastore cleanup-indexes. The command deletes all indexes for the production Cloud Datastore instance that are not mentioned in the local version of index.yaml.

The table shows concepts in Cloud Datastore that you should know from a relational database. Key differences between Cloud Datastore and a relational database are that Datastore is designed to automatically scale to a very large data set. Allowing applications to maintain high performance as they receive more traffic. Datastore maintains high performance by writing to 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 results set, as opposed to the data set. So a query whose results set contains 100 entities performs the same whether it searches over 100 entities or 1 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.

Cloud datastore