GAE: Datastore

Ing. Stabili Dario
dario.stabili@unimore.it
http://weblab.ing.unimo.it/people/stabili

Outline

- Introduction to Datastore
- Key Concepts
- Storing Data
- Queries
- Costs

- GAE Python Example

Online Material

- Rich online documentation and tutorials

- Source material for most of these slides:
  https://cloud.google.com/appengine/docs/python/ndb/

Introduction

GAE Datastore

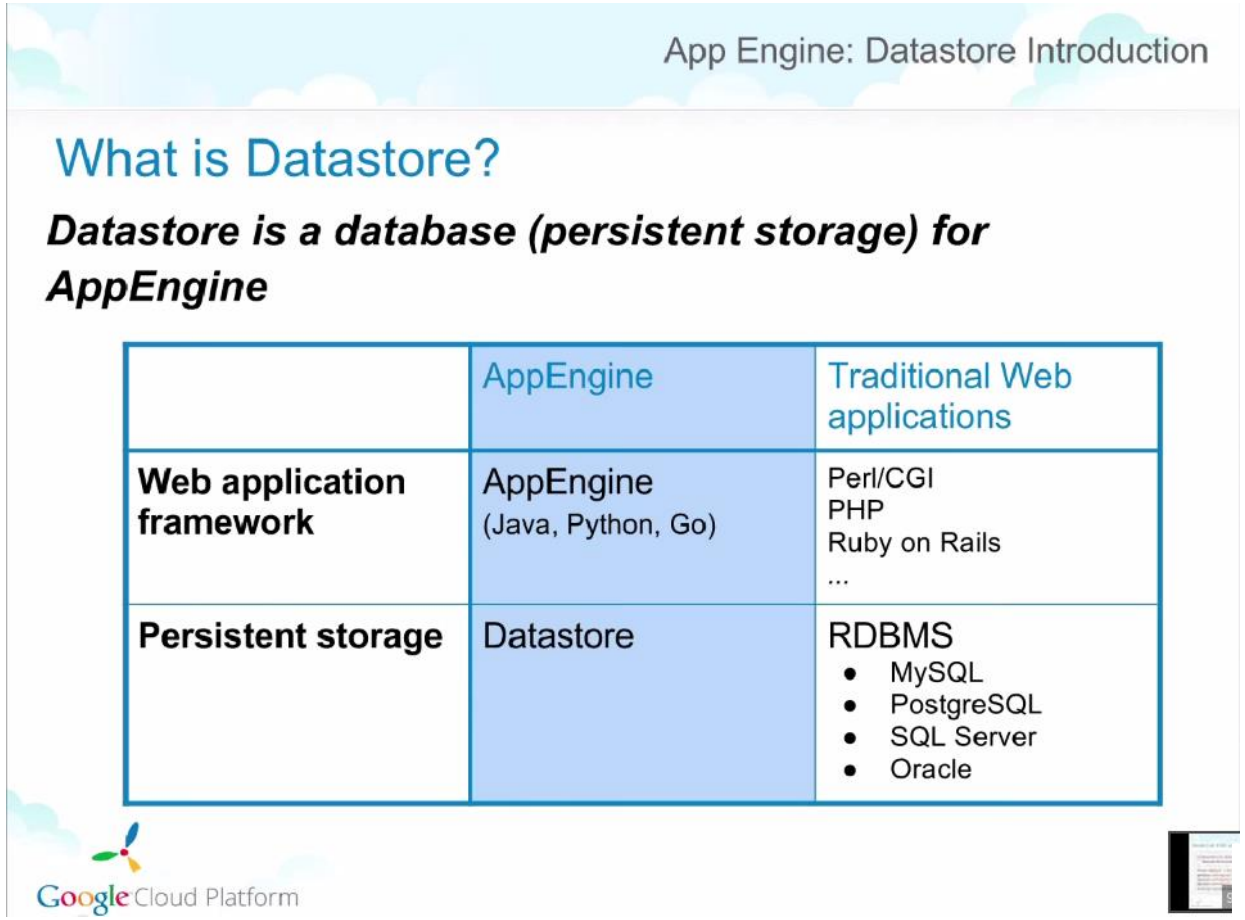
What is Datastore?

- **Datastore** is a database (**persistent storage**) for Google AppEngine

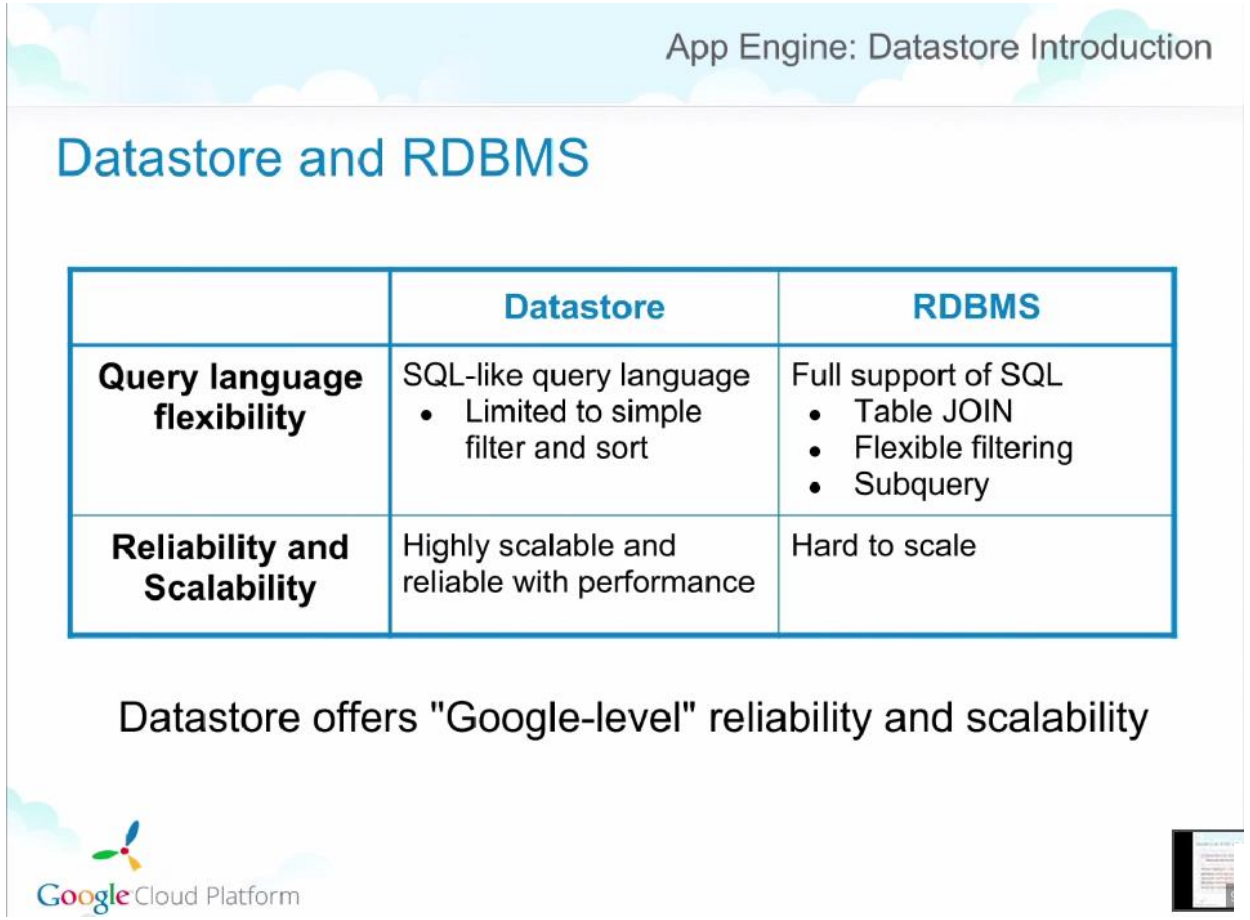
- Non-relational (**NoSQL**) database
  - ORM: Object-Relational Mapping
  - Motivation: scalability

- Built on top of **Google BigTable**

Introduction



Comparison with RDBMS



Scalability and Reliability

Problems of Scalability and Reliability

Single Instance

- Performance limited by machine resource
- Single point of failure

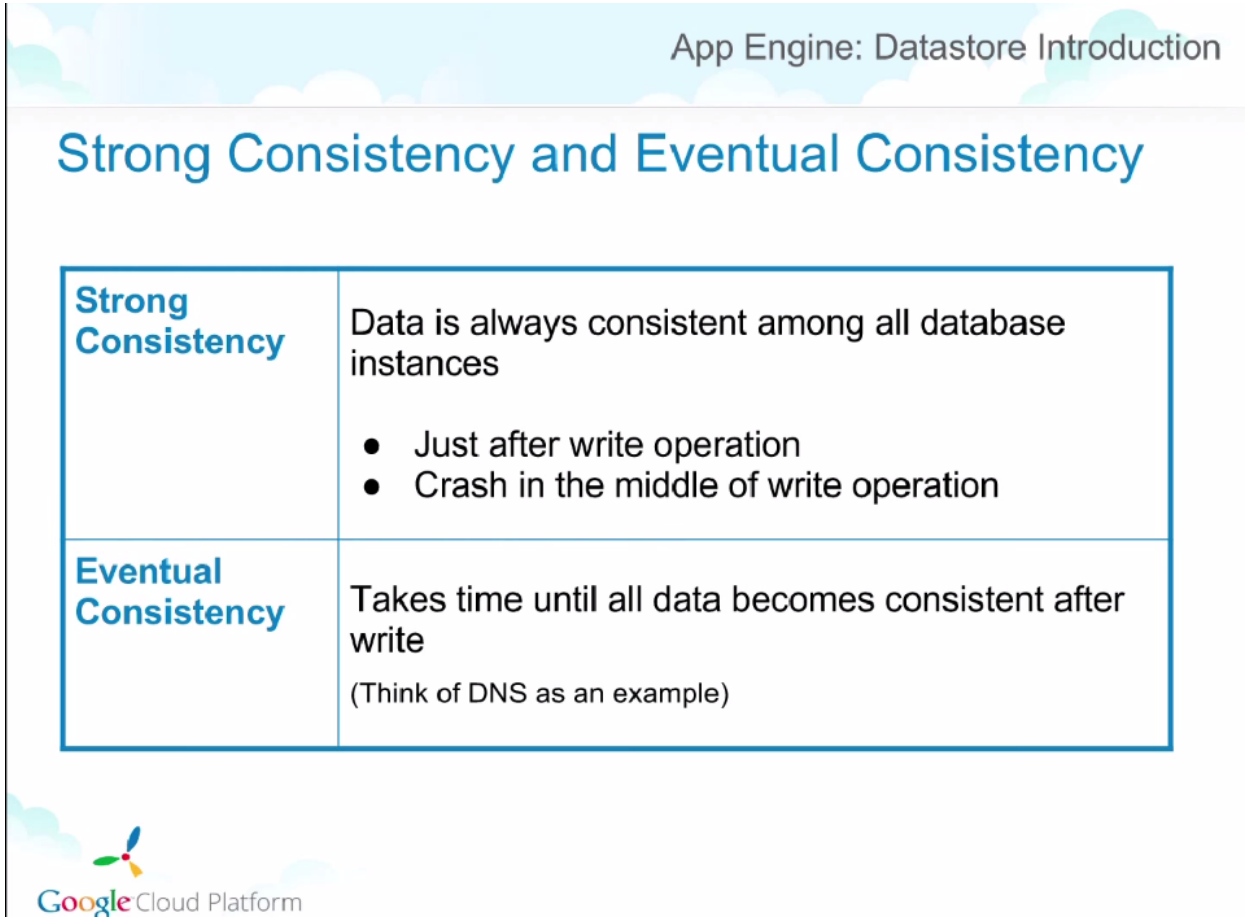
Replication (copy)

- Consistency among instances

Sharding (split among machines)

- Lock control (transaction)

Consistency



Key Concepts

GAE Datastore

Key Concepts

- Model
- Entity
- Property
- Key
- Kind

- Index
- Entity Group
- Ancestor Query

Datastore vs. RDBMS

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

Models

- A **model** is a class that describes a type of **entity**.
- A model acts as a sort of **database schema**: it describes a **kind** of data in terms of the properties.
  - It's roughly analogous to a **SQL Table**.
- An application uses ndb to define **models**.
- The underlying Datastore is very flexible in how it stores data objects—two entities of the same kind can have different properties.
  - An application can use NDB models to enforce type-checking but doesn't have to.

Models: Example

```
# Roughly analogous to defining a table “Greeting”
class Greeting(ndb.Model):
        # Roughly a string column “content”
        content = ndb.StringProperty()
        # Roughly a datetime column “date”
        date = ndb.DateTimeProperty(auto_now_add=True)
```

Entities and Properties

- NDB saves data objects, known as **entities**.
- An **entity** has one or more **properties** – values of one of several supported data types.
- For example, a **property** can be a string, an integer, or a reference to another entity.

Entities and Properties: Example

```
# Create new entity of kind Greeting
g = Greeting()

# Specify the content parameter
g.content = ‘Comment number 01’

# Store into the Datastore
# (more details afterwards)
g.put()
```

Complex Properties

- A model can have more **complex properties**.
  - **Repeated** properties are list-like.
  - **Structured** properties are object-like.
  - **Read-only computed** properties are defined via functions; this makes it easy to define a property in terms of one or more other properties.

Some Property Types

Property type	Description
`IntegerProperty`	64-bit signed integer
`FloatProperty`	Double-precision floating-point number
`BooleanProperty`	Boolean
`StringProperty`	Unicode string; up to 1500 bytes, indexed
`TextProperty`	Unicode string; unlimited length, not indexed
`BlobProperty`	Uninterpreted byte string: if you set **`indexed=True`**, up to 1500 bytes, indexed; if **`indexed`** is **`False`** (the default), unlimited length, not indexed. Optional keyword argument: **`compressed`**.
`DateTimeProperty`	Date and time (see Date and Time Properties)
`DateProperty`	Date (see Date and Time Properties)
`TimeProperty`	Time (see Date and Time Properties)

More on:
https://cloud.google.com/appengine/docs/standard/python/ndb/entity-property-reference

Property Options

Argument	Type	Default	Description
`indexed`	`bool`	Usually `True`	Include property in Cloud Datastore's indexes; if `False`, values cannot be queried but writes are faster. Not all property types support indexing; setting `indexed` to `True` fails for these. Unindexed properties cost fewer write ops than indexed properties.
`repeated`	`bool`	`False`	Property value is a Python list containing values of the underlying type (see Repeated Properties). Cannot be combined with `required=True` or `default=True`.
`required`	`bool`	`False`	Property must have a value specified.
`default`	Property's underlying type	None	Default value of property if none explicitly specified.
`choices`	List of values of underlying type	`None`	Optional list of allowable values.
`validator`	Function	`None`	Optional function to validate and possibly coerce the value. Will be called with arguments (*prop*, *value*) and should either return the (possibly coerced) value or raise an exception. Calling the function again on a coerced value should not modify the value further. (For example, returning `value.strip()` or `value.lower()` is fine, but not `value + '$'`.) May also return `None`, which means "no change". See also Writing Property Subclasses
`verbose_name`	string	`None`	Optional HTML label to use in web form frameworks like jinja2.

Key

- Each entity is identified by a **key**, an identifier unique within the application's datastore.
- The key can have a **parent**, another key.
- A key is instantiated as a **kind-ID** pair.
- This parent can itself have a parent, and so on; at the top of this "chain" of parents is a key with no parent, called the **root**.

Entity Group (EG)

- Entities whose keys have the **same root** form an **entity group** or **group**.

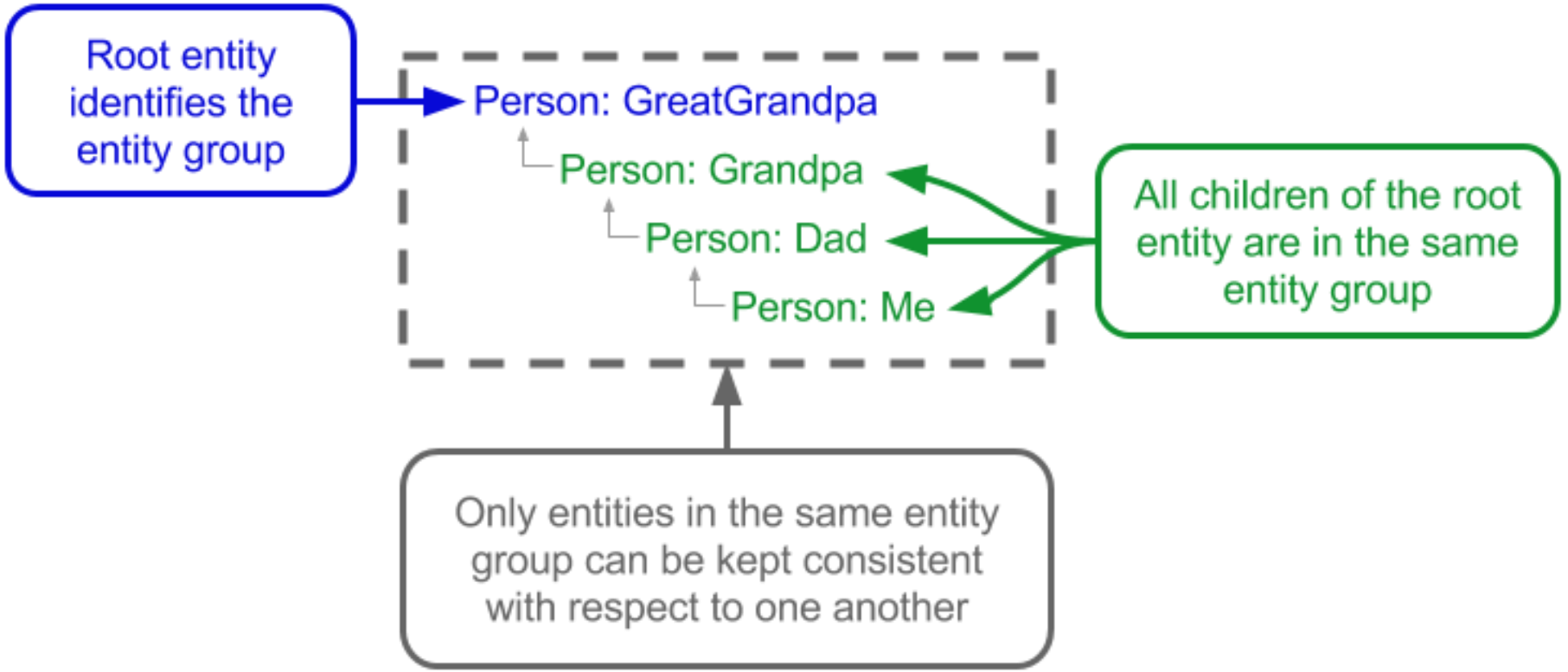
- If entities are in **different groups**, then changes to those entities might sometimes seem to occur "**out of order**".

- If the entities are unrelated in your application's semantics, that's fine. But if some entities' changes should be **consistent**, your application should make them part of the same group when creating them.

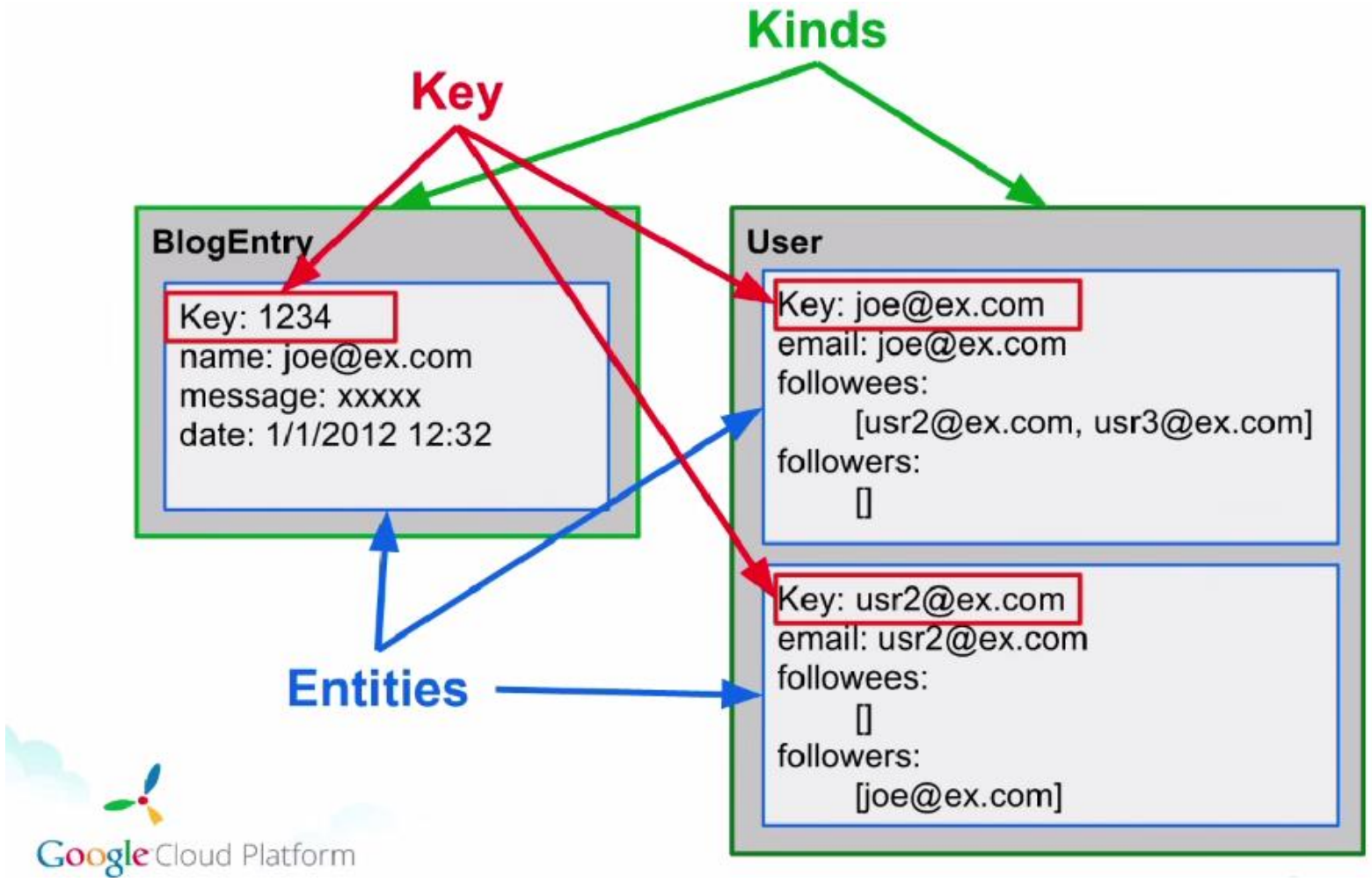
Entity Group: Example

```
# Create a parent key
parent_key = ndb.Key(Greeting, ‘MiaStringa’)

# Create two Entities within the same EG
g1 = Greeting(parent=parent_key, content=‘Commento 1’)
g2 = Greeting(parent=parent_key, content=‘Commento 2’)

# Store into the Datastore
# (more details afterwards)
g1.put()
g2.put()
```

Another Example



Kind, Entity and Key



Google Cloud Platform

Properties and Data Types

Each entity has one or more **named properties**

- Variety of data types (int, float, boolean, String, Date, etc.)
- Can be multi-valued

Datastore vs. RDBMS (2)

- Some differences:
  - An entity of a given kind is not required to have the same properties as other entities of the same kind
  - An entity can have a property of the same name as another entity, but with different type of values
  - An entity can have multiple values for a single property
  - Queries limited to simple filters and sorting

Storing Data

GAE Datastore

Storing Data

- An entity can be **created** by calling the model's class constructor and then stored by calling the `put()` method.

- Let’s start by looking at an example

Storing Data: Example

```
class Greeting(ndb.Model):
        content = ndb.StringProperty()
        date = ndb.DateTimeProperty(auto_now_add=True)

@app.route('/submitform',methods=['POST'])
class SubmitForm():
    # We set the parent key on each 'Greeting' to ensure each
    # guestbook's greetings are in the same entity group.
    guestbook_name = request.args.get('guestbook_name')
    greeting = Greeting(parent=ndb.Key(Greeting, guestbook_name or "*notitle*"),
                        content = request.args.get('content'))
    greeting.put()
```

Storing Data: Example

- This sample code defines the model class **Greeting**. Each Greeting entity has two properties: the text content of the greeting and the date the greeting was created.

- To create and store a new greeting, the application creates a new Greeting object and calls its `put()` method.

- To make sure that greetings in a guestbook don't appear "out of order" the application sets a parent key when creating a new Greeting. Thus, the new greeting will be in the same **entity group** as other greetings in the same guestbook. The application uses this fact when querying: it uses an ancestor query.

Insert/Update limits

- Within a same Entity Group (EG):
  - **1 update/sec** (rule of thumb suggested by Google)

- **Motivation**:
  - Updates within an Entity Group guarantee **strong consistency**
  - (More details afterwards)

- (Almost) **No limits** for entities belonging to different EGs

Queries

GAE Datastore

Queries

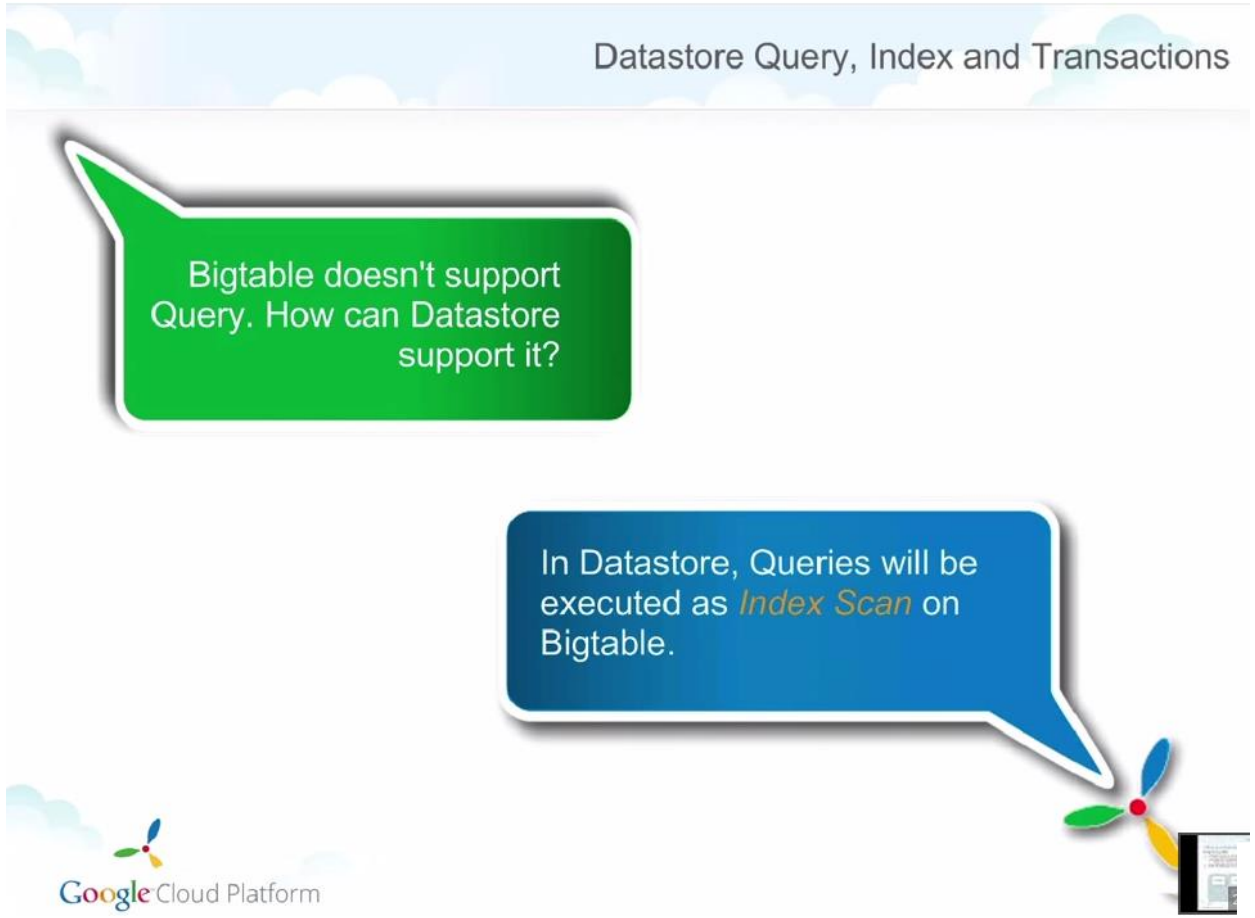
- An application can use **queries** to search the Datastore for entities that match specific search criteria called **filters**.

- Example: an application that keeps track of several guestbooks could use a query to retrieve **the last 10** messages from one guestbook, **ordered by date**.

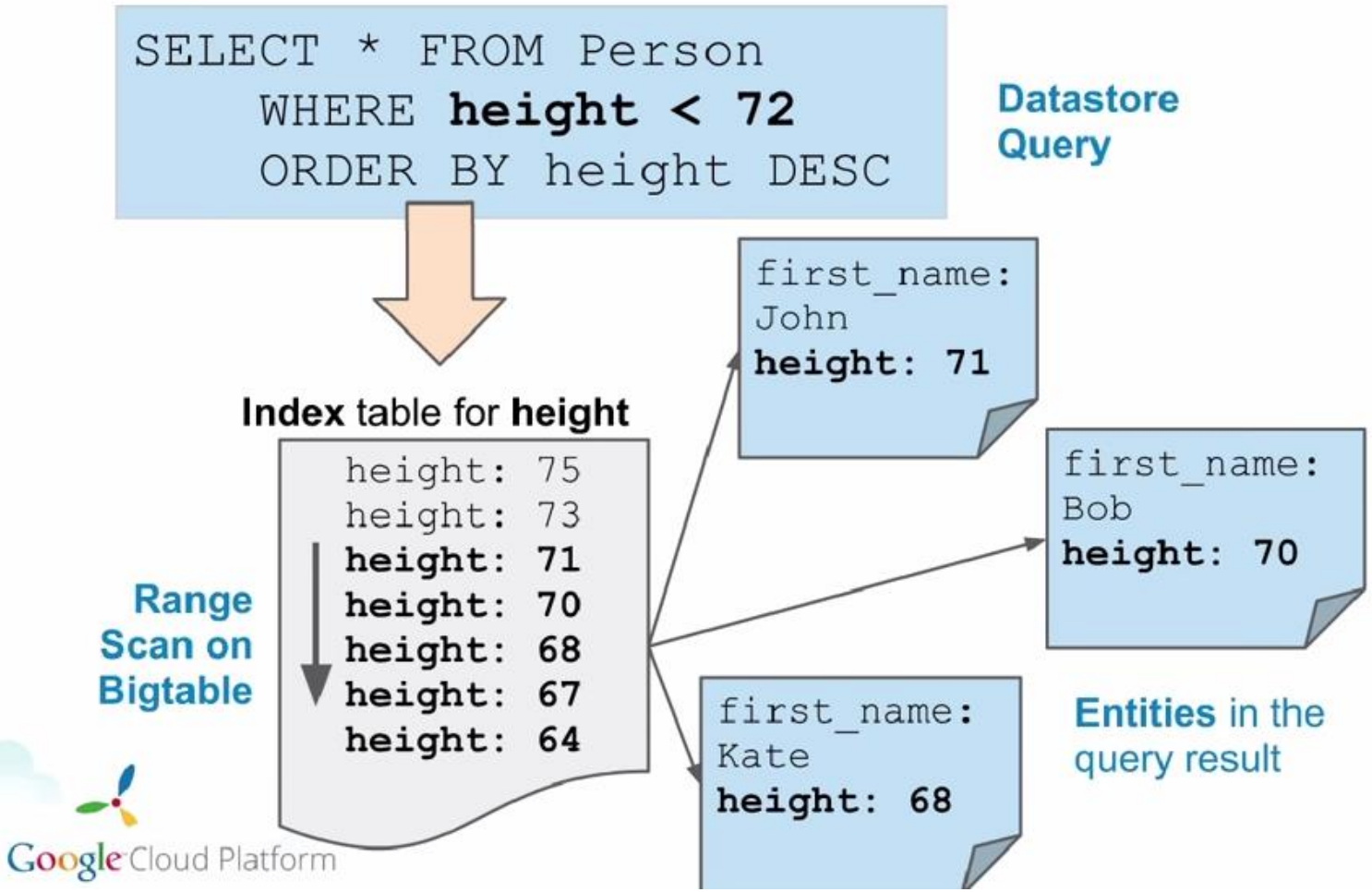
Queries: Example

```
def guestbook_key(id):
    """Constructs a Datastore key for a Guestbook entity.
    We use guestbook_name as the key.
    """
    return ndb.Key(Guestbook, id)

class Guestbook(ndb.Model):
    name = ndb.StringProperty()

class Greeting(ndb.Model):
    content = ndb.StringProperty()
    date = ndb.DateTimeProperty(auto_now_add=True)



greetings_query = Greeting.query(ancestor=guestbook).order(-Greeting.date)
greetings = greetings_query.fetch(10)
```

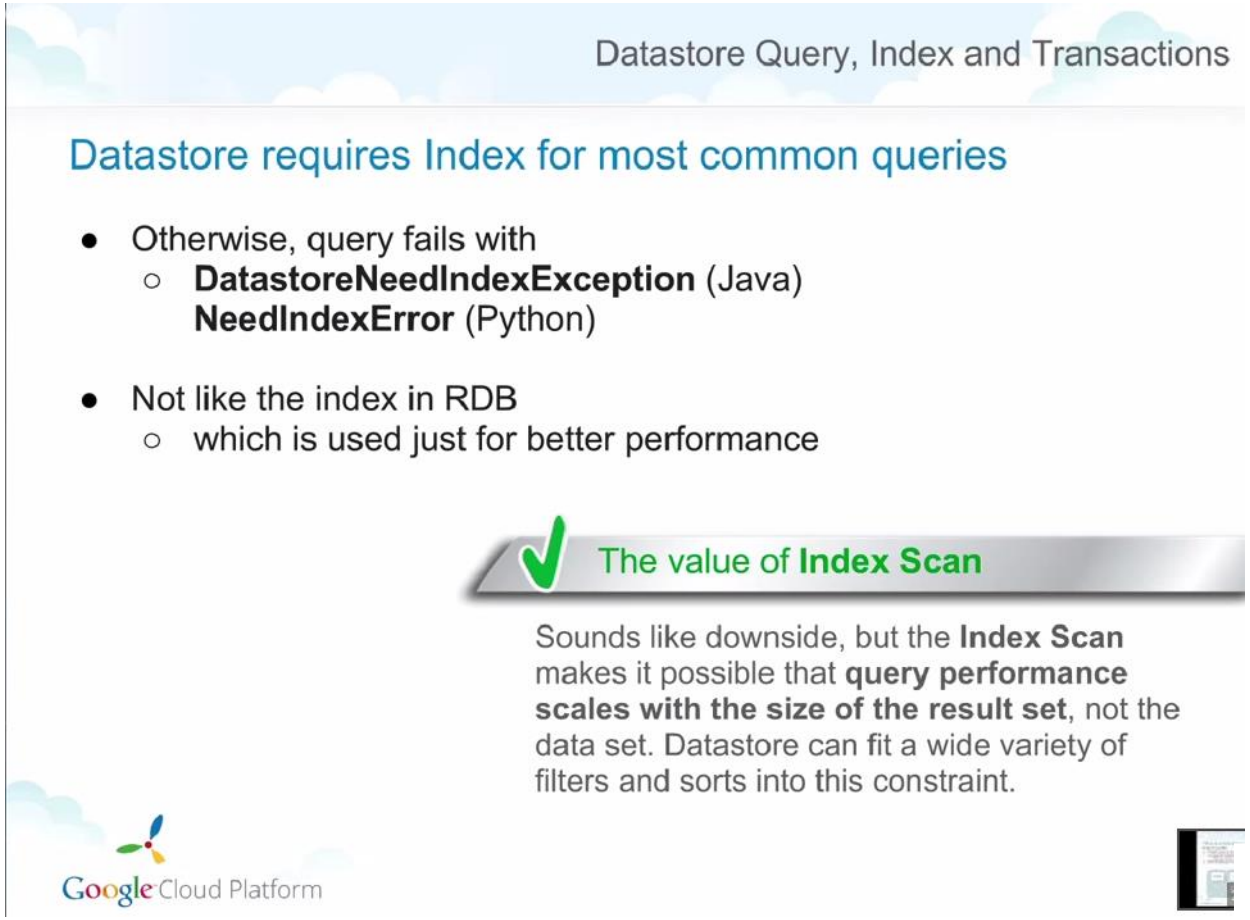
How can Datastore support Queries?



Query is Executed as Index Scan



Index Requirement



Queries and Indexes (1)

- A typical NDB query filters entities by **kind**.
- A query can also specify **filters** on entity property **values** and **keys**.
- A query can specify an **ancestor**, finding only entities that "belong to" some ancestor.
- A query can specify **sort order**.

Queries and Indexes (2)

- Some queries are more complex than others; the datastore needs **pre-built indexes** for these.
- These pre-built indexes are specified in a configuration file, `index.yaml`.
- On the **development server**, if you run a query that needs an index that you haven't specified, the development server automatically adds it to its **index.yaml**.
- But in your web site, a query that needs a not-yet-specified index fails.

Index.yaml file

```
indexes:

# AUTOGENERATED

# This index.yaml is automatically updated whenever the dev_appserver
# detects that a new type of query is run.  If you want to manage the
# index.yaml file manually, remove the above marker line (the line
# saying "# AUTOGENERATED").  If you want to manage some indexes
# manually, move them above the marker line.  The index.yaml file is
# automatically uploaded to the admin console when you next deploy
# your application using appcfg.py.

- kind: Greeting
  ancestor: yes
  properties:
  - name: date
    direction: desc
```

Indexes

- You can **tune indexes** manually by editing the file before uploading the application.

- This index mechanism supports a wide range of queries and is suitable for most applications.

- However, it does not support some kinds of queries common in other database technologies. In particular, **JOINs** aren't supported. (but you can use workarounds to simulate them)

Filtering by Property Values

- Usually you don't want to retrieve all entities of a given kind; you want only those with a specific value or range of values for some property.

```
query = Account.query(Account.userid == 42)
```

Supported Operations

- NDB supports these operations:
  - property == value
  - property < value
  - property <= value
  - property > value
  - property >= value
  - property != value
  - property.IN([value1, value2])

AND and OR operations

```
query = Article.query(
    ndb.AND(
        Article.tags == 'python',
        ndb.OR(
                Article.tags.IN(['ruby', 'jruby']),
                ndb.AND(
                        Article.tags == 'php',
                        Article.tags != 'perl'
                )
        )
    )
)
```

Sort Orders

- You can use the `order()` method to specify the order in which a query returns its results.

- This method takes a list of arguments, each of which is either a property object (to be sorted in ascending order) or its negation (denoting descending order). For example:

```
query = Greeting.query().order(Greeting.message, -Greeting.userid)
```

Ancestor Queries (1)

- Ancestor queries allow you to make **strongly consistent queries** to the datastore
- However entities with the same ancestor are **limited to 1 write per second**.

Non-ancestor example:

```
class Customer(ndb.Model):
    name = ndb.StringProperty()

class Purchase(ndb.Model):
    customer = ndb.KeyProperty(kind=Customer)
    price = ndb.IntegerProperty
```

- To find all the purchases that belong to the customer:

```
Purchase.query(customer=customer_entity.key).fetch()
```

Ancestor Queries (2)

- **Ancestor Query** example:

```
class Customer(ndb.Model):
    name = ndb.StringProperty()

class Purchase(ndb.Model):
    price = ndb.IntegerProperty()
```

- Each purchase has a key, and the customer has its own key as well. However, **each purchase key will have the customer_entity's key embedded in it**. Remember, this will be limited to one write per ancestor per second. The following creates an entity with an ancestor:

```
purchase1 = Purchase(parent=customer_entity.key)
```

- To query for the purchases of a given **customer**, use the following query.

```
Purchase.query(ancestor=customer_entity.key).fetch()
```

GQL

- GQL is a **SQL-like language** for retrieving entities or keys from the App Engine Datastore.

- While GQL's features are different from those of a query language for a traditional relational database, the GQL syntax is similar to that of SQL.

- The **GQL syntax** is described in the GQL Reference.

- Example:

```
query = ndb.gql("SELECT * FROM Account WHERE spam > :1", 10)
```

Costs

GAE Datastore

Costs for Operations

Costs for Datastore Calls

Datastore operations are billed as follows:

Operation	Cost
Read / Write	$0.06 per 100,000 operations
Small	Free

Calls to the datastore API result in the following billable operations. Small datastore operations include calls to allocate datastore ids or keys-only queries. These operations are free. This table shows how calls map to datastore operations:

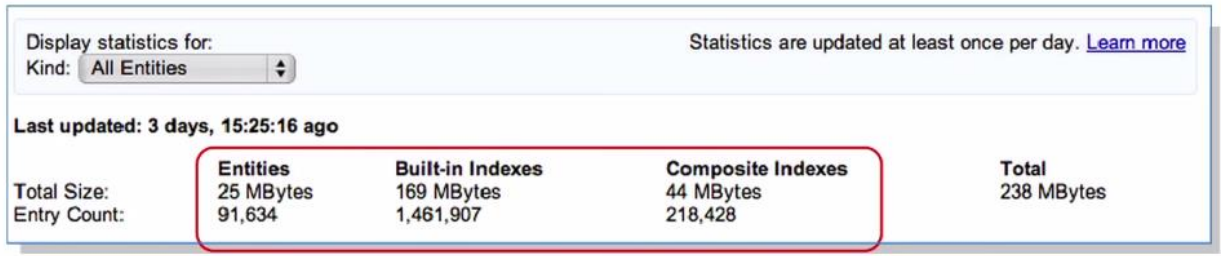
API Call	Datastore Operations
Entity Get (per entity)	1 read
New Entity Put (per entity, regardless of entity size)	2 writes + 2 writes per indexed property value + 1 write per composite index value
Existing Entity Put (per entity)	1 write + 4 writes per modified indexed property value + 2 writes per modified composite index value
Entity Delete (per entity)	2 writes + 2 writes per indexed property value + 1 write per composite index value
Query*	1 read + 1 read per entity retrieved
Projection Query	1 read
Projection Query with "Distinct" option	1 read + 1 read per entity retrieved

* Queries that specify an offset are charged for the number of results that are skipped in addition to those that are returned.

Cost of Index

Cost of Index

- Most of the time, **Index** will consume a certain part of **Datastore space** as well as **instance hour**
- Need to take the **cost of Index** into account for **cost estimation**
  - See Understanding Write Costs for details
- If you add new Index on a large set of entities, it may take **long time** (sometimes, several days!)



Google Cloud Platform

Quotas and Limits

Quotas and Limits

Various aspects of your application's Datastore usage are counted toward your resource quotas:

- Data sent to the Datastore by the application counts toward the **Data Sent to Datastore API** quota.
- Data received by the application from the Datastore counts toward the **Data Received from Datastore API** quota.
- The total amount of data currently stored in the Datastore for the application cannot exceed the **Stored Data (billable)** quota. This includes all entity properties and keys, as well as the indexes needed to support querying those entities. See the article How Entities and Indexes Are Stored for a complete breakdown of the metadata required to store entities and indexes at the Bigtable level.

For information on systemwide safety limits, see the Quotas and Limits page and the **Quota Details** section of the Administration Console. In addition to such systemwide limits, the following limits apply specifically to the use of the Datastore:

Limit	Amount
Maximum entity size	1 megabyte
Maximum transaction size	10 megabytes
Maximum number of index entries for an entity	20000
Maximum number of bytes in composite indexes for an entity	2 megabytes