# Datastore Queries and Indexes

## Google App Engine

# Agenda

1. Queries
2. Indexes
3. Restrictions and Workarounds for Queries
4. Cost of Indexing

# Introduction to Queries

Google Cloud Platform

# Query

- Query by **kind**
- Filter on **property** or **parent**
- Results can be **sorted**

**Consistency Model**

Queries always return results from the **INDEX**

- Eventually Consistent
  - Queries without an ancestor filter get results from the last index update
- Strongly Consistent
  - Queries using an ancestor filter forces applicable index updates to complete
  - get() forces the index to update before returning the results

# Filters

Filter on

- property values
- keys
- ancestors

Use an Ancestor Filter for Strong Consistency

Filter on property values

- Equality filter (Equal to)
- IN -- Member of a list
- Inequality filters
  - not equal to
  - less than
  - less than or equal to
  - greater than
  - greater than or equal to

# Query

- Adding multiple filters to a query

## Java

```
Query q = new Query("Person");
Query.Filter filter1 = new FilterPredicate(...);
Query.Filter filter2 = new FilterPredicate(...);
Query.Filter comboFilter = CompositeFilterOperator.and(filter1, filter2);
q.setFilter(comboFilter);
```

## Python

```
q = Person.all()
q.filter("name =", "John")
q.filter("sport =", "Sailing")
```

# Sort

- Query results can be sorted by property
  - Sort by **ascending** or **descending** value of a property
  - Some restrictions on sorting (discussed later)

## Java

```
q.addSort("name");
```

## Python

```
q.order('name')
```

# Indexes

# Index

Bigtable doesn't support Query. How can Datastore support it?

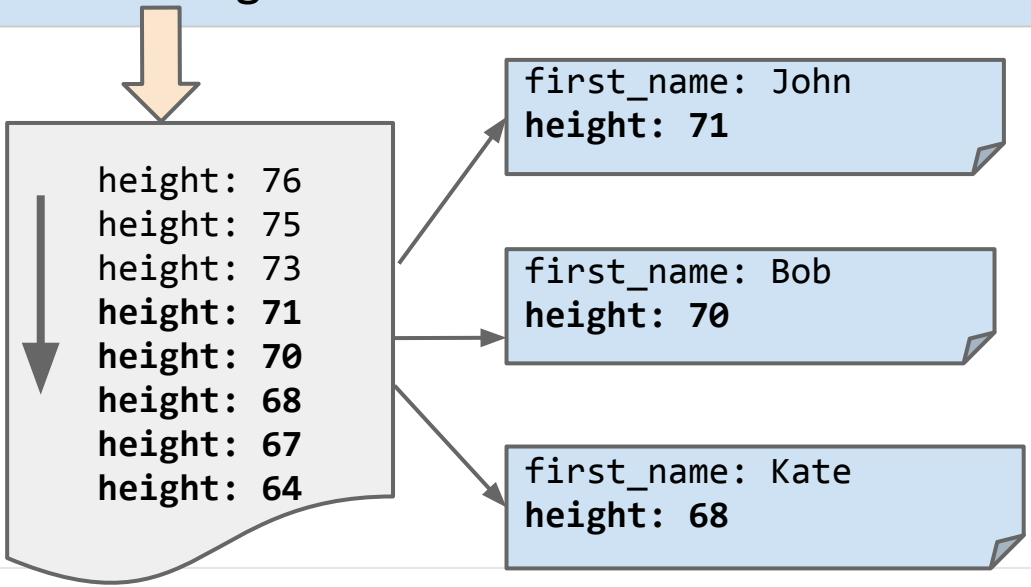
In Datastore, Queries are executed as *Index Scans* on Bigtable.

# Index

- Queries are executed as index scans

```
SELECT * FROM Person
    WHERE height < 72
    ORDER BY height DESC
```

**Datastore Query**



# Index

Datastore requires **index** for ALL queries, otherwise the **query fails**

↳ Not like the index in RDBMS, which is used to improve performance

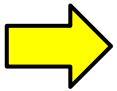

The **Index Scan** makes it possible for **query performance to scale with the size of the result set**, not the data set.

# Index

Single-property index

- Query for kind = Person, first_name = ‘Ben’
- Scan with prefix [Person first_name Ben]

Single-property index is created automatically



| Kind | Property | Value | Entity Key |
|---|---|---|---|
| Person | first_name | Alice | EntityKey112 |
| Person | first_name | Ben | EntityKey122 |
| Person | first_name | Bridgit | EntityKey223 |
| Person | first_name | Cathy | EntityKey233 |

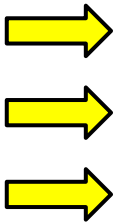
**Key of Index table** **Value**

# Index

Single-property index

- Query for kind: Person
  - first_name > = “A” and first_name < “C”
- Scan with prefix [Person first_name A, Person First_name B]

**Two** single-property indices are created automatically:
- ascending
- descending



| Kind | Property | Value | Entity Key |
|---|---|---|---|
| Person | first_name | Audrey | EntityKey112 |
| Person | first_name | Ben | EntityKey122 |
| Person | first_name | Bridgit | EntityKey223 |
| Person | first_name | Cathy | EntityKey233 |

**Key of Index table** **Value**

# Index

Single-property indexes can support queries with:

- Equality filters on one or more properties (merge join)

  **first_name = 'Bob' AND last_name = 'James'**

- Inequality filters on one property

  **first_name >= 'B' AND first_name < 'C'**
  **first_name != 'Bob'**

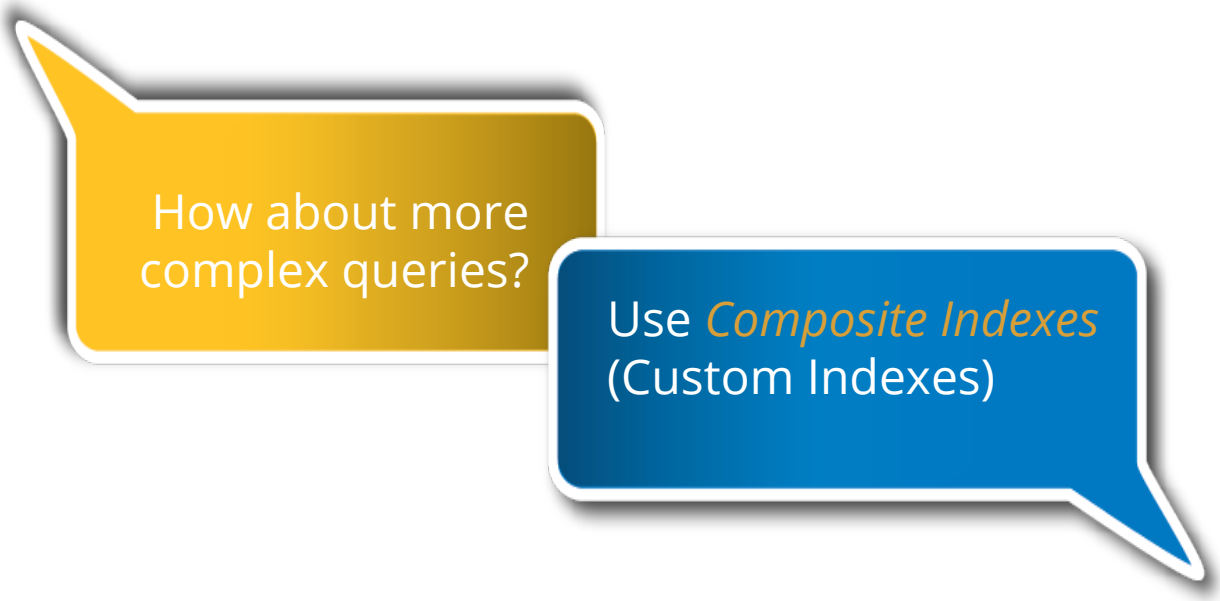
  will be executed as

  **first_name < 'Bob' OR first_name > 'Bob'**

- One sort order

  **ORDER BY last_name ASC**

# Index



# Index

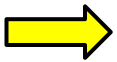
Composite index example

- Query for kind : Person

  last_name = “Smith”

  first_name > “A” and first_name < “D”

Equality filter + Inequality filter



| *Kind /* | *last_name /* | *first_name* |
|---|---|---|
| Person / | Raley / | Jane |
| Person / | Smith / | Ben |
| Person / | Smith / | Cathy |
| Person / | Smith / | Daniel |
| Person / | Thomas / | Alice |

Composite Index must be **explicitly configured**.

# Index

How to create indexes

- App Engine creates single property indexes for all properties
- You can run queries in the development server to create custom indexes
- You can create or edit index configuration file

# Index

Configuration files for composite index

**Java**

- XML

  WEB-INF/**datastore-indexes.xml**
  WEB-INF/appengine-generated/**datastore-indexes-auto.xml**

- yaml

  WEB-INF/**index.yaml**

**Python**

- **index.yaml**

Reference: Datastore Indexes (Java) (Python)

# Index

Index configuration example

- Query for kind : Person
  last_name = “Smith”
  first_name > “A” and first_name < “D”

| *Kind* | *last_name* | *first_name* |
|---|---|---|
| Person | Raley | Jane |
| Person | Smith | Ben |
| Person | Smith | Cathy |
| Person | Smith | Daniel |
| Person | Thomas | Alice |

index.yaml

```
- kind: Person
  properties:
  - name: last_name
    direction: asc
  - name: first_name
    direction: asc
```

# Index

Indexes for multi-valued properties

- An index entry is created for EVERY value of a property

Entity kind Person
  name = Brian
  lucky_number = {1, 5, 7, 9}

Kind / property / value

| |
|---|
| Person / lucky_number / 1 |
| Person / lucky_number / 5 |
| Person / lucky_number / 7 |
| Person / lucky_number / 9 |

# Index

Multi-valued properties in Queries

- Multi-valued properties match a query
  - IF AT LEAST ONE value matches ALL the filters

Entity kind Person
name = Brian
lucky_number = {1, 5, 7, 9}

Matches query for?
Kind is Person
lucky_number > 2 and
lucky_number < 6

OK

Matches query for?
Kind is Person
lucky_number = 1

OK

Matches query for?
Kind is Person
lucky_number > 1 and
lucky_number < 5

X

# Index

Look out for Combinatorial Explosion!

Entity kind Person
  name = Brian
  number = {1, 5, 7, 9}
  colors = {red, blue, green}
  friends = {anna, ben, cathy, ...zoe}

| name | numbers | colors | friends |
|---|---|---|---|
| Brian | 9 | red | anna |
| Brian | 9 | red | ben |
| Brian | 9 | red | cathy |
| ... | ... | ... | ... |
| Brian | 9 | red | zoe |

| name | numbers | colors | friends |
|---|---|---|---|
| Brian | 9 | blue | anna |
| Brian | 9 | blue | ben |
| Brian | 9 | blue | cathy |

| name | numbers | colors | friends |
|---|---|---|---|
| Brian | 9 | green | anna |
| Brian | 9 | green | ben |
| Brian | | | |
| ... | ... | | |
| Brian | 9 | green | zoe |

4 x 3 x 26 = 312 index entries

# Restrictions and Workarounds for Queries

Google Cloud Platform

# Query

Sounds like Datastore Queries are not as flexible as RDBMS queries.

True. It's very important to understand the **restrictions** of Datastore Query and workarounds for them *before* you start designing your data model.

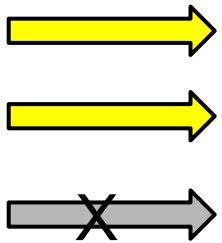
# Query: Missing Properties

- Entities with **no property or an** unindexed value are **not included in results**

Query for:

Kind = Person
last_name != Arundel

**Missing Property**
is not equal to **Null/None**



| Kind | last_name | first_name |
|---|---|---|
| Person | Anderson | Jane |
| Person | Artwood | |
| Person | | Jenny |

# Query: Inequality Filters

- Inequalities filters: limited to one property per query

Query for:

first_name = Cathy

last_name > Able

last_name < Mooney

OK

Query for:

first_name > Cathy

last_name > Able

X

# Query: Inequality Filters and Sorting

- A property with an inequality filter must be **sorted first**

Query for:

first_name = Cathy

last_name > Able

sort by last_name

OK

Query for:

last_name > Able

sort by first_name

X

# Query: Restrictions on Inequality Filters

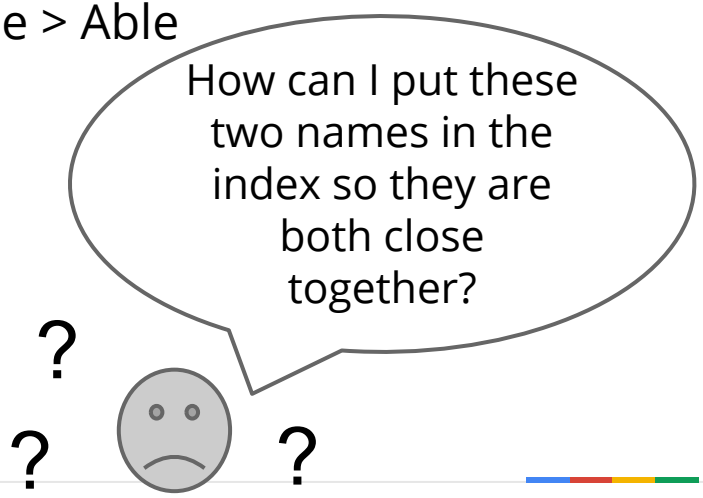
**Why** the restrictions on inequality filters ?

- To avoid having to scan the entire index table, the query mechanism relies on all of a query's potential results being able to be converted to a key range scan on the index

Just suppose we allowed...
first_name > Aaron
last_name > Able


Zoe Acton

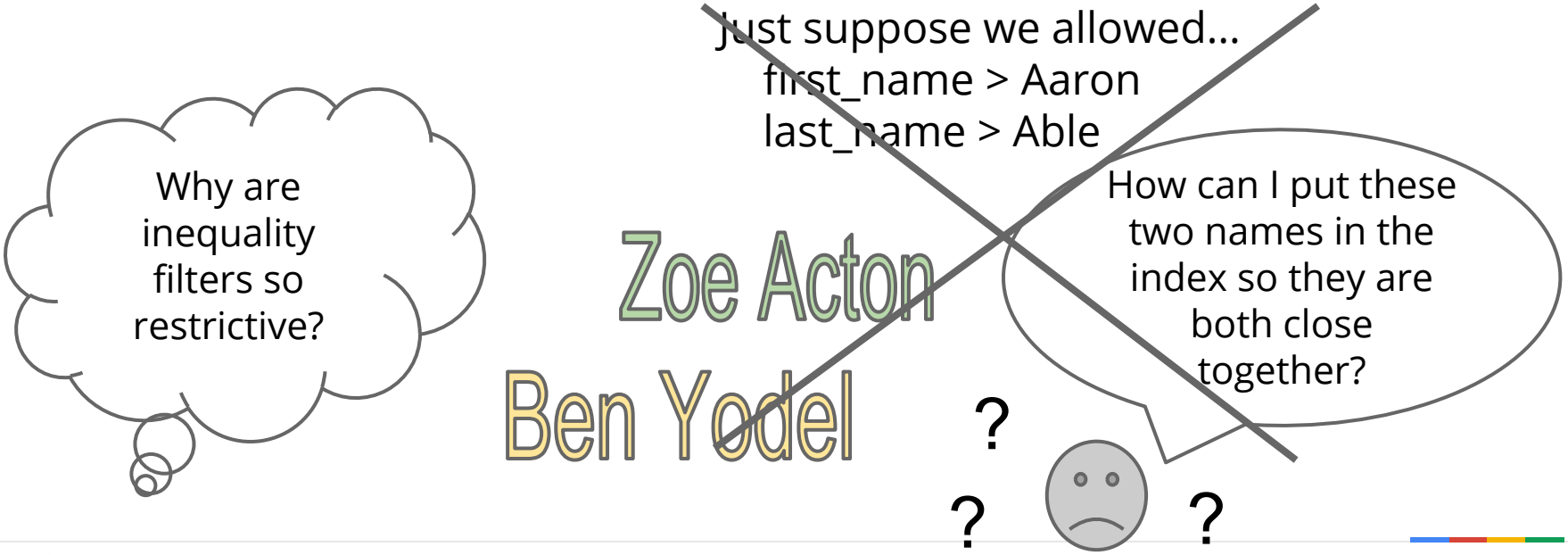
Ben Yodel



# Query: Restrictions on Inequality Filters

**Why** the restrictions on inequality filters ?

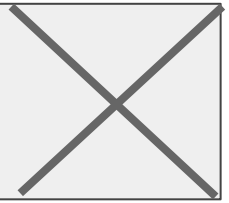
- To avoid having to scan the entire index table, the query mechanism relies on all of a query's potential results being able to be converted to a key range scan on the index



# Query: JOIN Operation

- JOIN operation is not supported
- Use **Denormalization**
  It's a known practice for any scalable database design

```
SELECT * FROM PERSON p, ADDRESS a
WHERE a.person_id = p.id AND p.age > 25
AND a.country = "US"
```



```
SELECT FROM Person
WHERE age > 25 and country = "US"
```

# Query: Aggregation

- Aggregation queries are not supported
- Datastore does not support aggregation queries

  (group by, having, sum, avg, max, min)
- Use a special entity that maintains aggregated values
  - Counter entity
  - Be careful not to make the entity bottleneck

    (by 1 updates/sec limit)
  - Use **Sharding Counter** pattern
  - Or **Memcache putIfUntouched() + Datastore insert**

# Query: Aggregation (continued)

- Use batch processing to aggregate values asynchronously
  - Backend instance
  - App Engine MapReduce
- Datastore Statistics
  - For counting entities
  - Updated once per day
- Use Sorting for MIN() or MAX()
  - Sort by a property: the first entity will have min/max value.

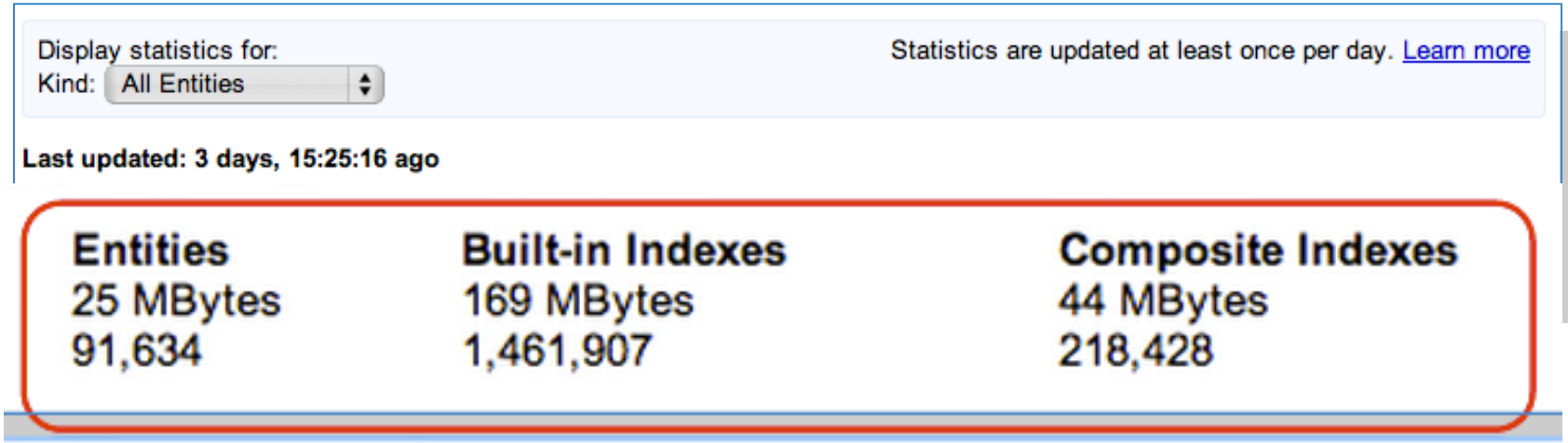
# Cost of Indexing

# Cost of Index

- **Index** consumes **Datastore space** & **instance hours**
- Take the **cost of Index** into account for **cost estimation**
  Read:
  - Understanding Write Costs
  - How Entities and Indexes are Stored
- New Index for a large set of entities may take a **long time**

# Datastore Statistics

- View statistics in the Admin Console
  - Datastore > Datastore Statistics



# Index : Cost

Setting a property without indexing

**Python**

```
class Conference (db.Model):
  main_contact = db.StringProperty(indexed=False)
```

**Java**

```
conference.setUnindexedProperty("mainContact", "Adam Bolivar");
```

➔ Don’t index long strings (use Search API)
➔ Text and Blob are automatically treated as unindexed

# Index : Cost

Deleting Indexes

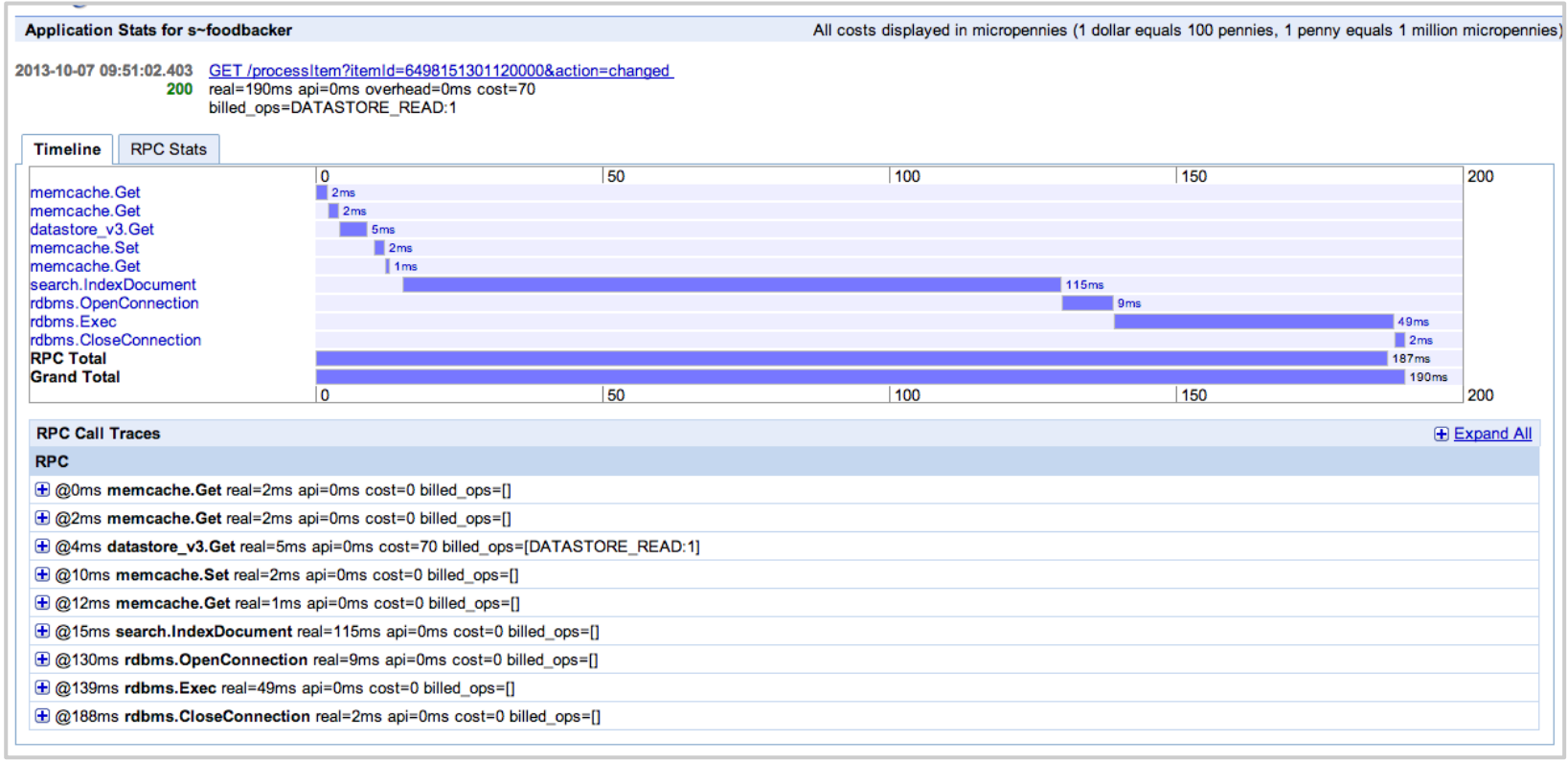
- Existing indexes remain when index config changes
- Lets you:
  - Leave an older version of the app running while new indexes are being built
  - Revert to the older version if needed

To delete all unused indexes:

- Update index config file
- Run:

```
appcfg.sh vaccum_indexes myapp
```

# Profiling with AppStats Tool



**Application Stats for s~foodbacker** — All costs displayed in micropennies (1 dollar equals 100 pennies, 1 penny equals 1 million micropennies)

2013-10-07 09:51:02.403 GET /processItem?itemId=6498151301120000&action=changed
200 real=190ms api=0ms overhead=0ms cost=70
billed_ops=DATASTORE_READ:1

Timeline | RPC Stats



**RPC Call Traces** — Expand All

| RPC |
|---|
| @0ms **memcache.Get** real=2ms api=0ms cost=0 billed_ops=[] |
| @2ms **memcache.Get** real=2ms api=0ms cost=0 billed_ops=[] |
| @4ms **datastore_v3.Get** real=5ms api=0ms cost=70 billed_ops=[DATASTORE_READ:1] |
| @10ms **memcache.Set** real=2ms api=0ms cost=0 billed_ops=[] |
| @12ms **memcache.Get** real=1ms api=0ms cost=0 billed_ops=[] |
| @15ms **search.IndexDocument** real=115ms api=0ms cost=0 billed_ops=[] |
| @130ms **rdbms.OpenConnection** real=9ms api=0ms cost=0 billed_ops=[] |
| @139ms **rdbms.Exec** real=49ms api=0ms cost=0 billed_ops=[] |
| @188ms **rdbms.CloseConnection** real=2ms api=0ms cost=0 billed_ops=[] |

# Enable AppStats Recording

## Java

web.xml

```
<filter>
  <filter-name>appstats</filter-name>
   <filter-class>com.google.appengine.tools.appstats.AppstatsFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>appstats</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

## Python

appengine_config.py

```
def webapp_add_wsgi_middleware(app):
    from google.appengine.ext.appstats import recording
    app = recording.appstats_wsgi_middleware(app)
    return app
```

# Enable AppStats Admin Console Page

## Java

web.xml

```
<servlet>
  <servlet-name>appstats</servlet-name>
  <servlet-class>com.google.appengine.tools.appstats.AppstatsServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>appstats</servlet-name>
  <url-pattern>/appstats/*</url-pattern>
</servlet-mapping>
```

## Python

app.py

```
builtins:
- appstats: on
```

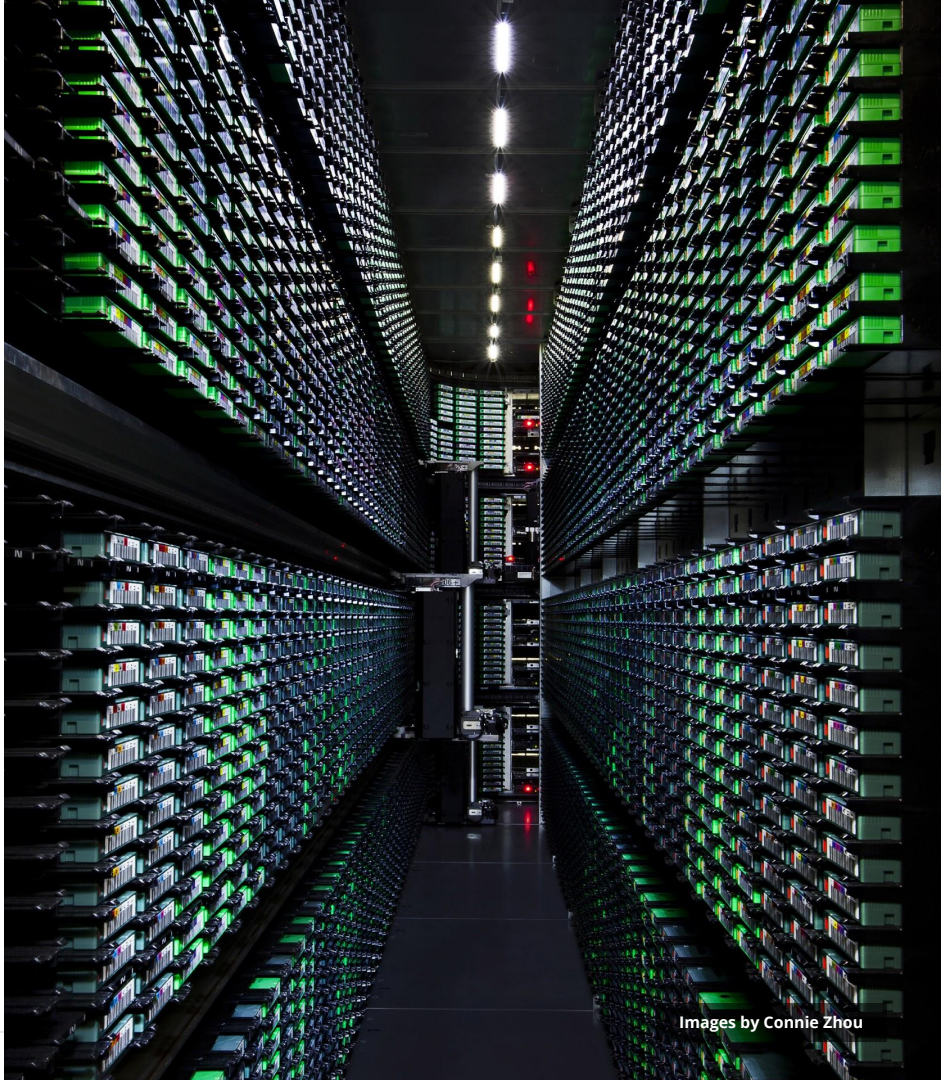
# Quiz

Which of the following methods can be used to create datastore indexes?
*(select **3** of the available options)*

- ❑ App Engine automatically creates single property indexes for all properties
- ❑ You need to manually run queries in the development server to create single property indexes
- ❑ You can run queries in the development server to create custom indexes
- ❑ App Engine automatically creates custom indexes when deploying your application
- ❑ You can edit index configuration files manually
- ❑ None of the above

# Codelab

- Try out multiple queries in the same source code as the previous Hands on exercise



# Resources

- **Developer Documentation**
  - Datastore Queries (Java) (Python)
  - Transactions (Java) (Python)

- **App Engine Datastore Articles**
  - Mastering the Datastore series
  - Transaction Isolation
  - Handling Datastore Errors
  - Understanding Write Costs
  - How Entities and Indexes are Stored

# Resources

- **Google I/O Sessions**
  - Under the Covers of the Google App Engine Datastore, Ryan Barret, Google I/O 2008
  - Building Scalable, complex Apps on App Engine, Brett Slatkin, Google I/O 2009
  - Next gen queries, Alfred Fuller, Google I/O 2010


cloud.google.com
Images by Connie Zhou