Google™

Under the Hood of the App Engine Datastore

Joe Gregorio
Developer Advocate
Google App Engine

Questions

http://tinyurl.com/ete-hood

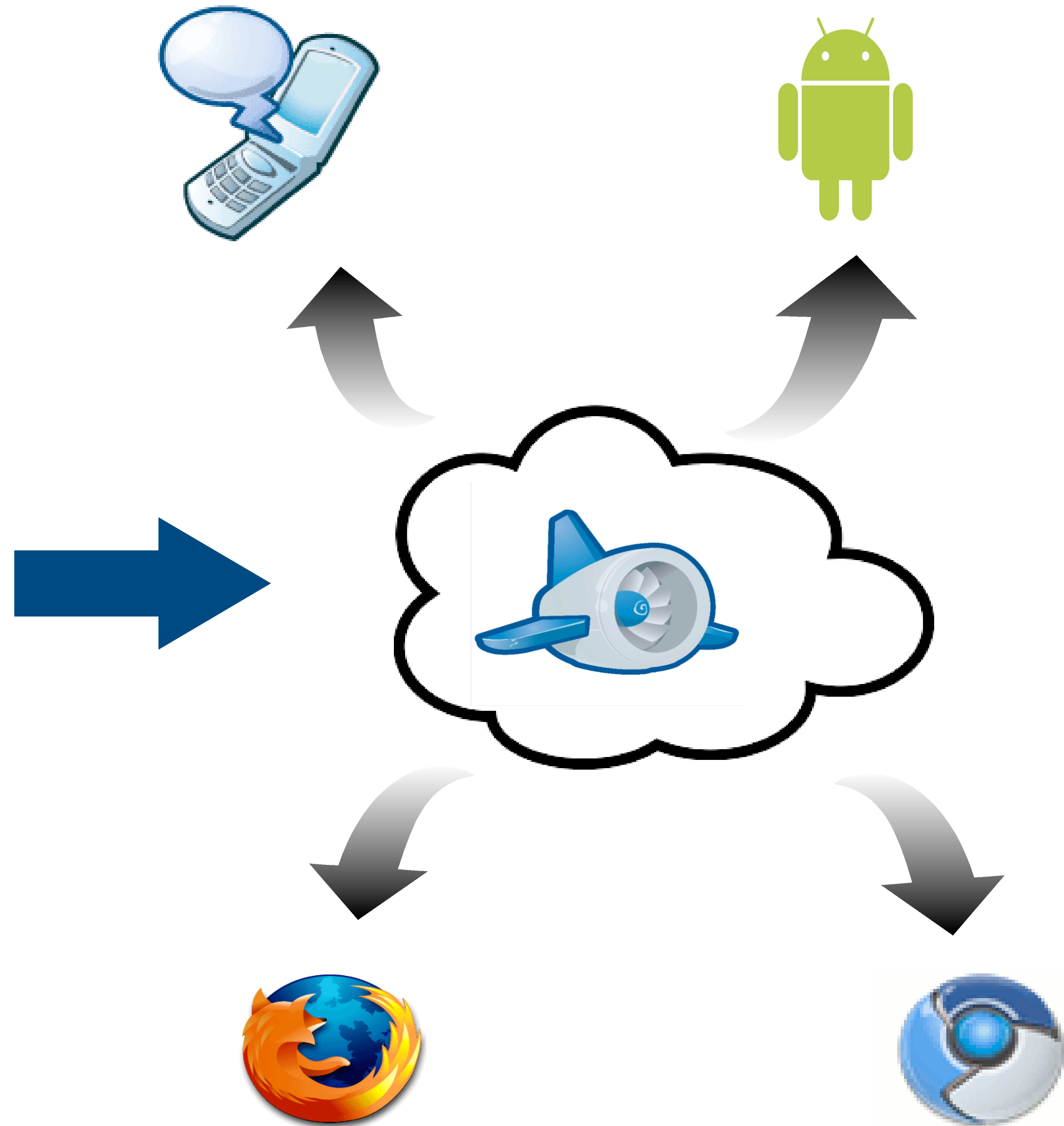
Or feel free to ask as we go along

Building Web Apps

The challenges

What is Google App Engine

main.py index.yaml app.yaml



Challenges building web apps

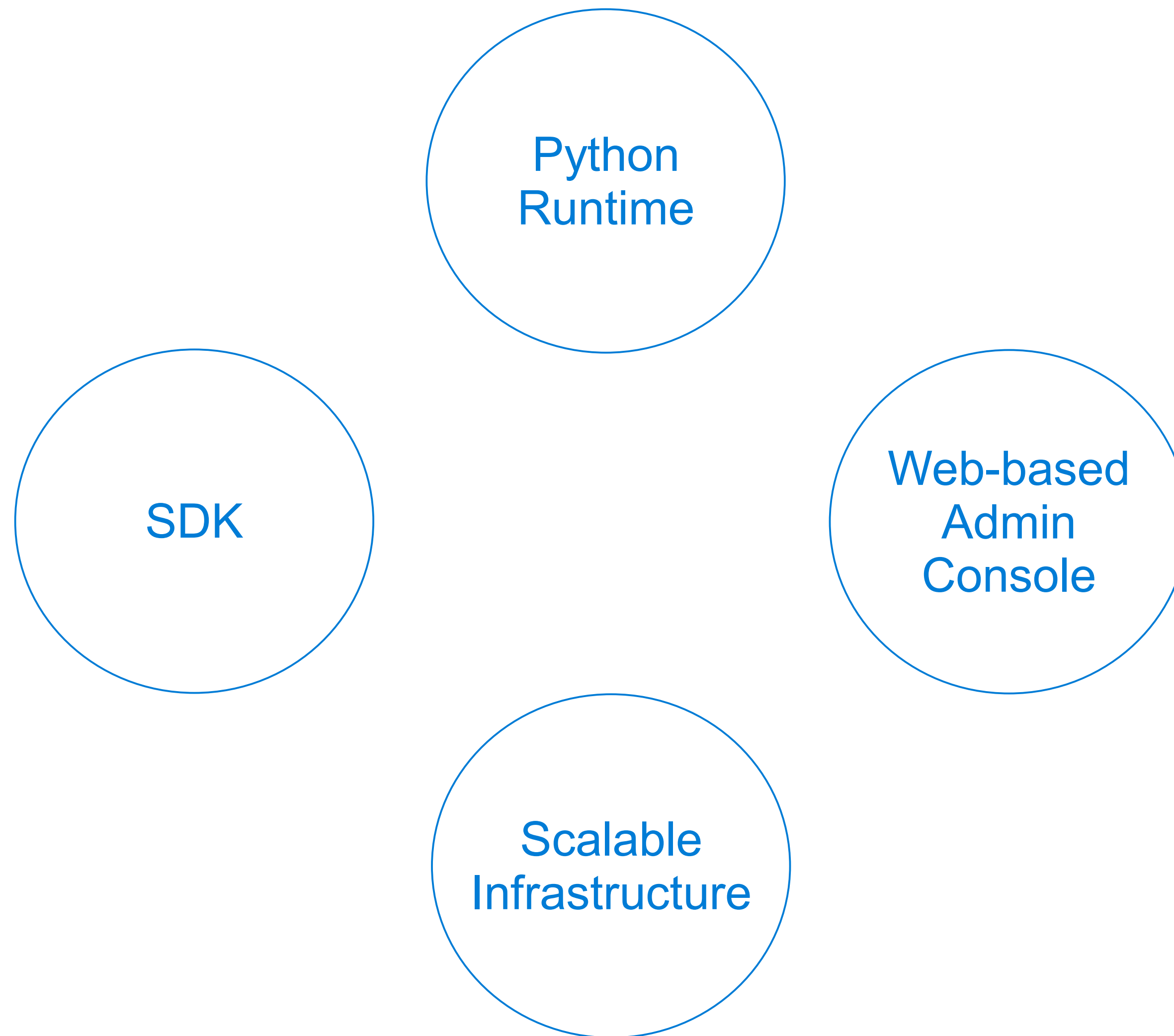
Easy to start

Easy to scale

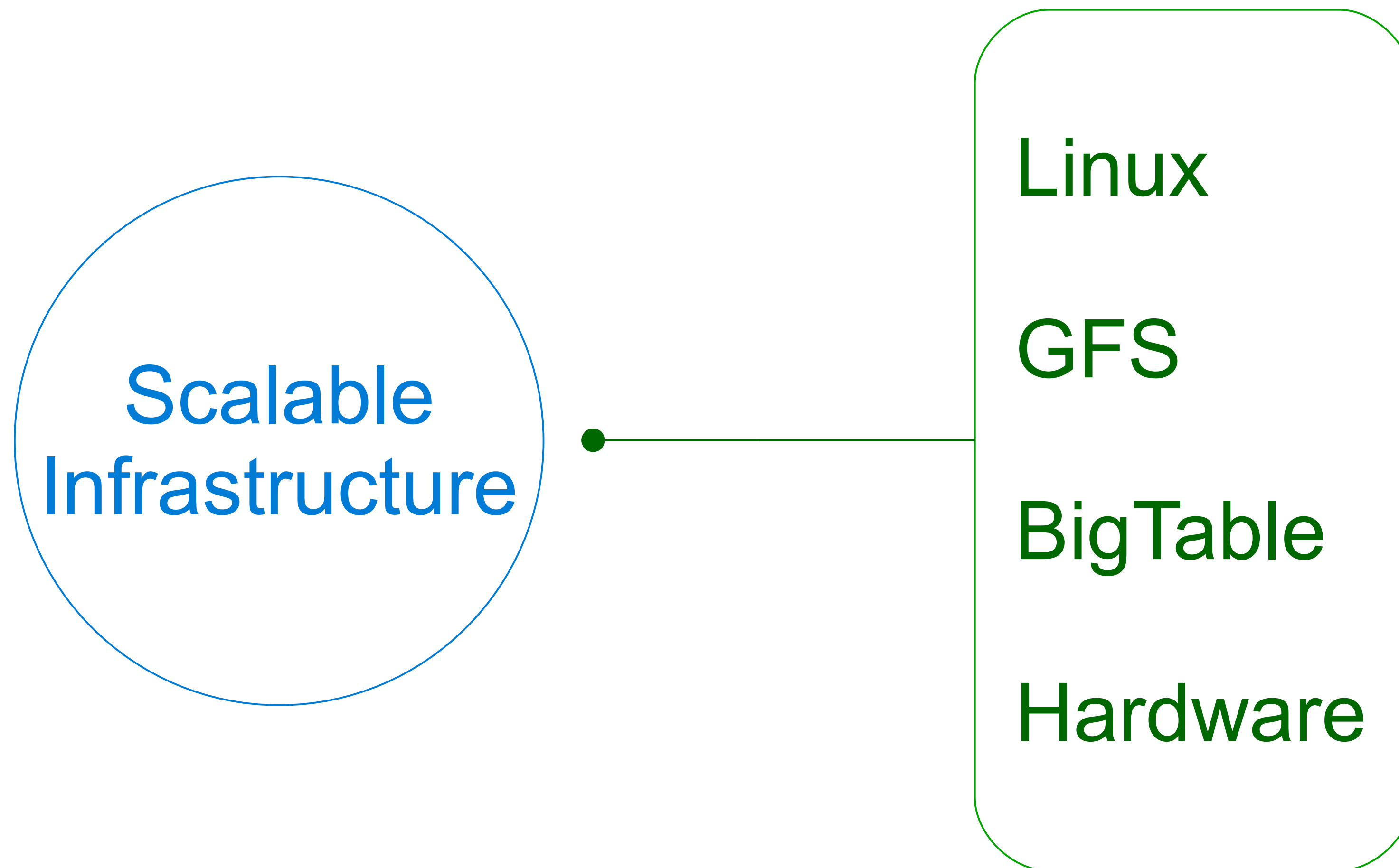
The Components

The pieces of Google App Engine

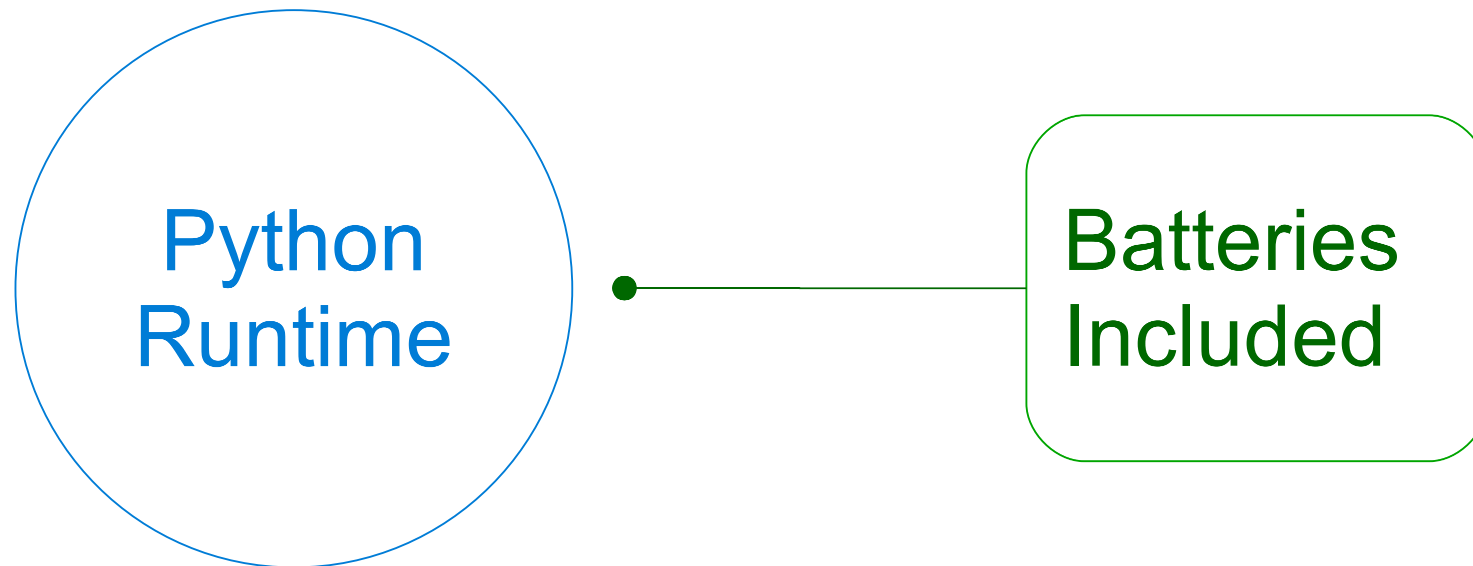
Components



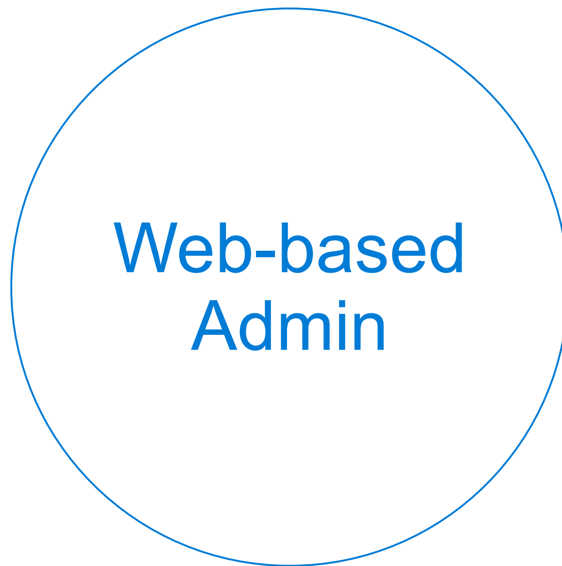
Components



Components



Components



Google App Engine

project-playground Version: 1

Dashboard

Logs

Datastore

- Indexes
- Data Viewer

Administration

- Application Settings
- Developers
- Versions
- Admin Logs

Resources

- Documentation
- Developer forum
- Downloads

Charts

Requests/Second | all | 24 hr

1.00
0.800
0.600
0.400
0.200

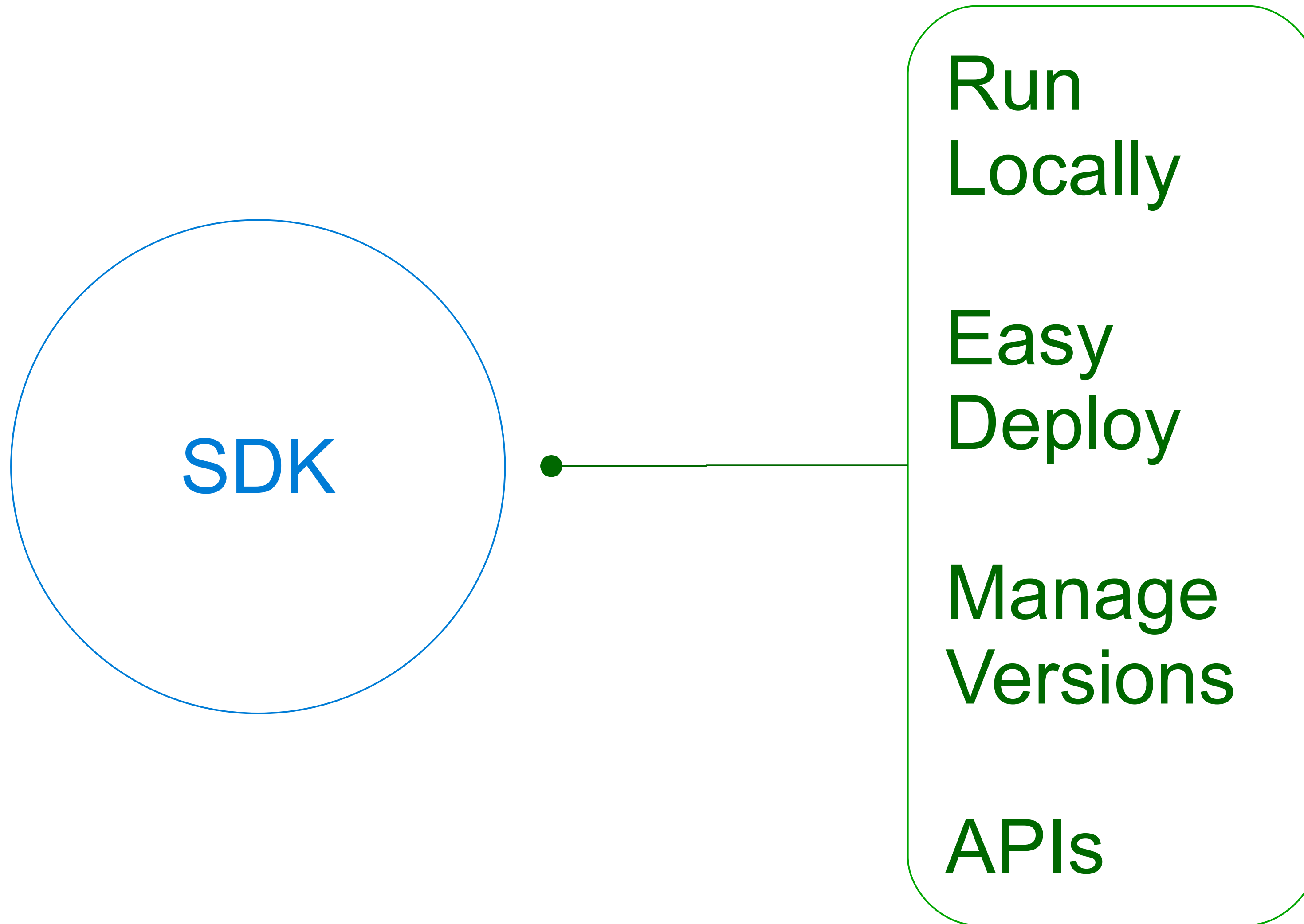
Application Quotas

CPU Used
Data Sent
Data Received
Emails Sent
Megabytes Stored

Current Load

URI

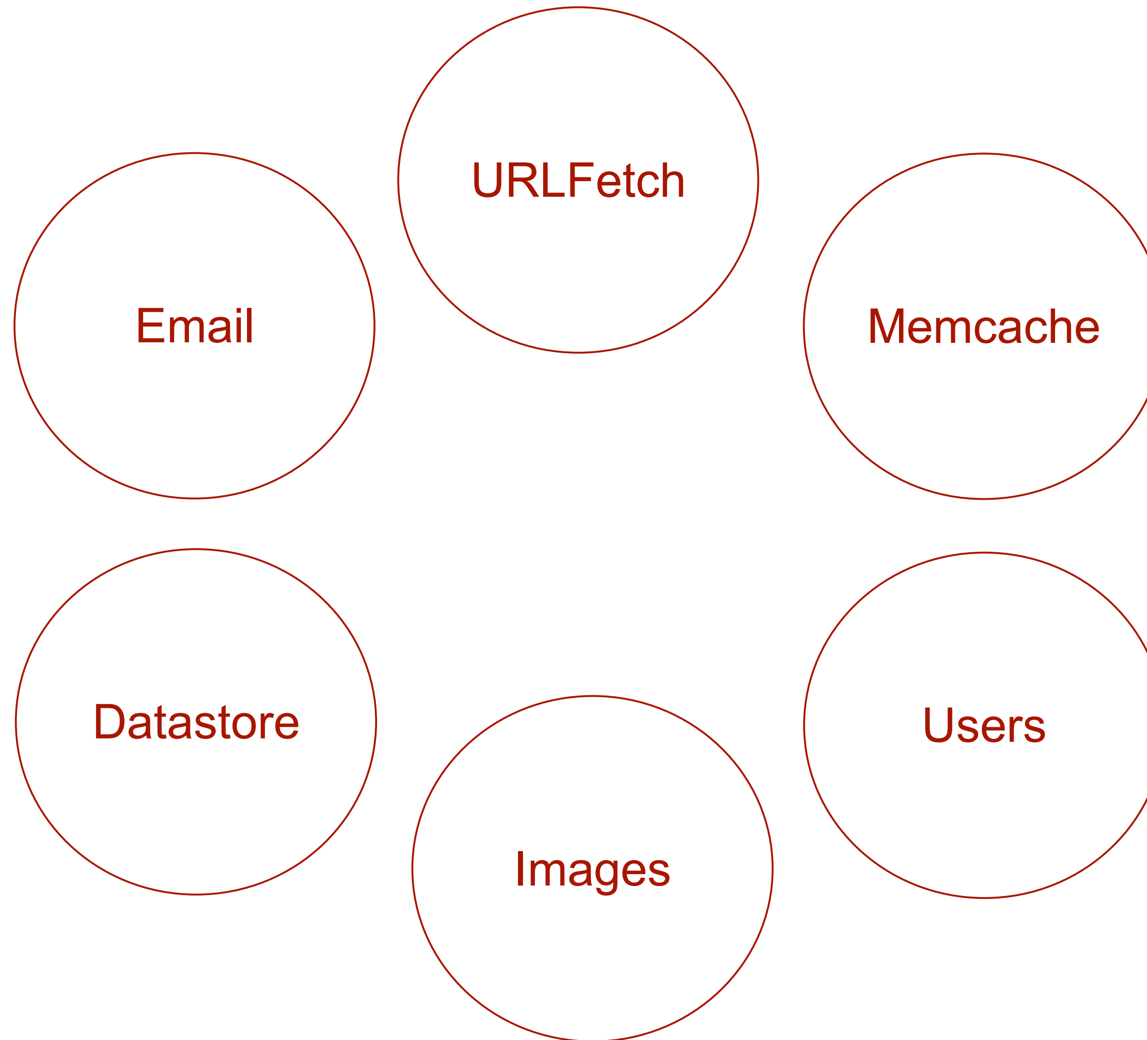
Components



APIs

Software Development Kit

APIs



Current Status

Current Status

Always free to get started

- 1 GB storage
- 1 GB/day bandwidth
- ~5 M pageviews/month

Purchasing additional resources

Outgoing Bandwidth	$0.12/GB
Incoming Bandwidth	$0.10/GB
CPU Time	$0.10/hr
Stored Data	$0.15/GB/month
Recipients Emailed	$0.0001

Future

- More languages
- XMPP
- Large upload/download support
- Offline processing

The Datastore

Looking under the hood

Google

Constraints Precede Performance

db.Model

Bigtable

Entity Table

Queries and Indexes
- Kind
- Single-property
- Composite
- Merge Join

Entity Groups and Transactions

db.Model

Overview

```
from google.appengine.ext import db

class ToDo(db.Model):
  description = db.StringProperty()
  created = db.DateTimeProperty(auto_now_add=True)


# Add a new ToDo item to the datastore
item = ToDo(description=”Get Milk”)
item.put()


# Query for ToDo items in reverse chron order
items = ToDo.all().order("-created").fetch(100)
for item in items:
   print item.description
```

Subclass from db.Model

```
from google.appengine.ext import db

class ToDo(db.Model):
  description = db.StringProperty()
  created = db.DateTimeProperty(auto_now_add=True)


# Add a new ToDo item to the datastore
item = ToDo(description=”Get Milk”)
item.put()


# Query for ToDo items in reverse chron order
items = ToDo.all().order("-created").fetch(100)
for item in items:
   print item.description
```

Create and store an instance to the datastore

```
from google.appengine.ext import db

class ToDo(db.Model):
  description = db.StringProperty()
  created = db.DateTimeProperty(auto_now_add=True)


# Add a new ToDo item to the datastore
item = ToDo(description=”Get Milk”)
item.put()


# Query for ToDo items in reverse chron order
items = ToDo.all().order("-created").fetch(100)
for item in items:
   print item.description
```

Query for stored entities

```
from google.appengine.ext import db

class ToDo(db.Model):
  description = db.StringProperty()
  created = db.DateTimeProperty(auto_now_add=True)


# Add a new ToDo item to the datastore
item = ToDo(description=”Get Milk”)
item.put()


# Query for ToDo items in reverse chron order
items = ToDo.all().order("-created").fetch(100)
for item in items:
   print item.description
```

Exact same query but now in GQL

```
from google.appengine.ext import db

class ToDo(db.Model):
  description = db.StringProperty()
  created = db.DateTimeProperty(auto_now_add=True)


# Add a new ToDo item to the datastore
item = ToDo(description=”Get Milk”)
item.put()


# Query for ToDo items in reverse chron order
items = db.GqlQuery(
  “SELECT * from ToDo ORDER BY created DESC"
  ).fetch(100)
for item in items:
   print item.description
```

BigTable

What the datastore is built on

Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

http://research.google.com/archive/bigtable-osdi06.pdf

Bigtable

Row Name	Columns
a	...
b	...
c	...
d	...
f	...
j	...
k	...
l	...
n	...

A Sharded, Sorted Array

Two kinds of tables

Entity Table

Key	Entity
a	...
b	...
c	...

Index Table

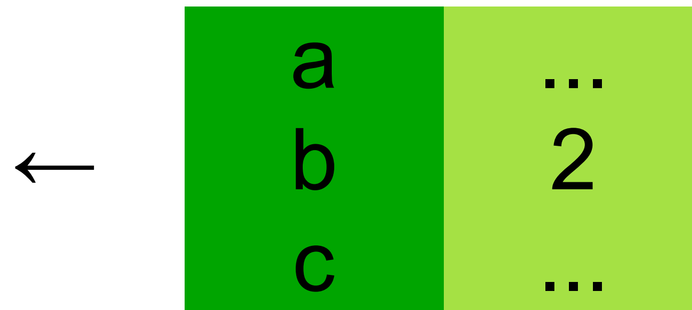
Index
foo
bar
baz

One entity table for all entities in every application stored in App Engine

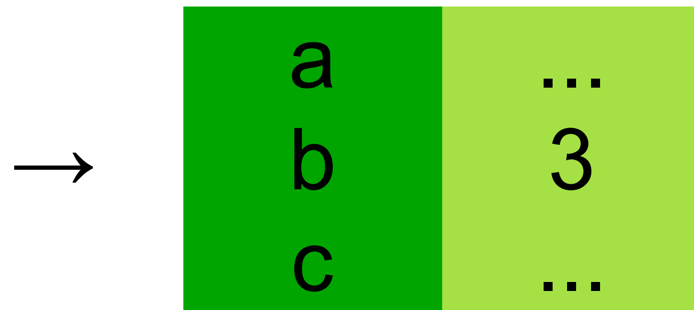
Bigtable Capabilities

- Read
- Write
- Delete
- Single-row transaction
- Scan
  - prefix
  - range

Read



Write

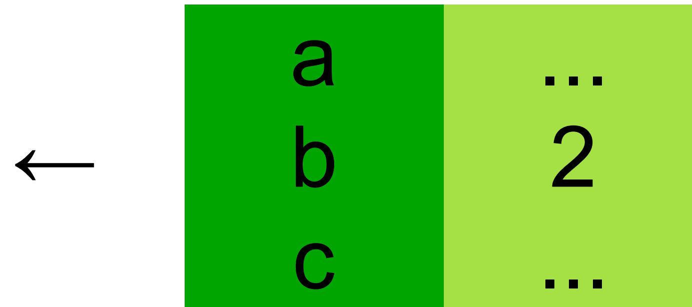


Delete

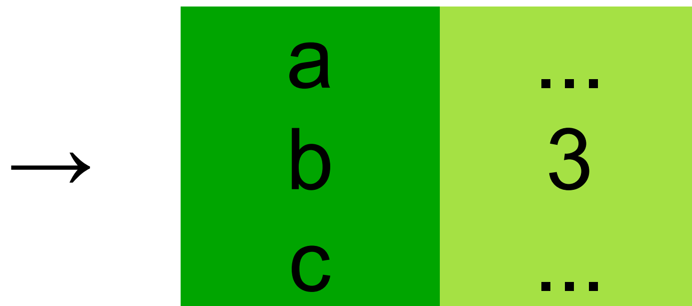
x

a	...
b	3
c	...

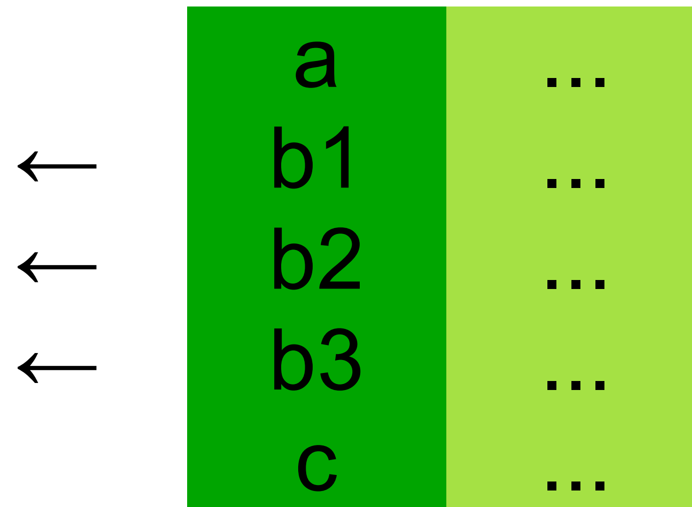
Transactions



Read, Modify, Write

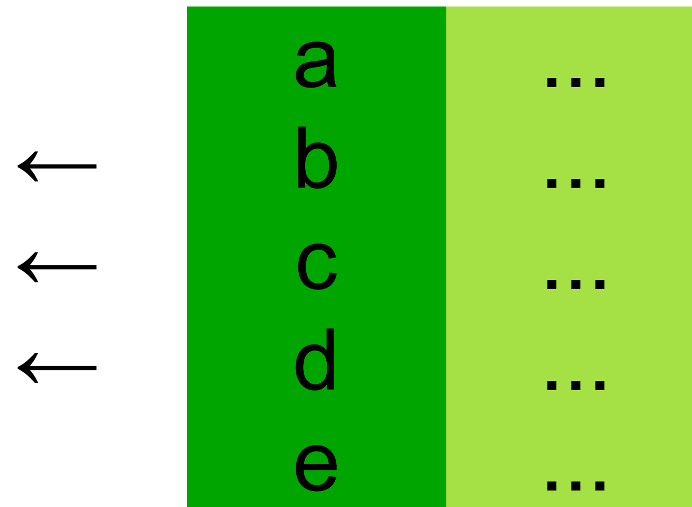


Prefix Scan



Scan: Prefix 'b'

Range Scan



Scan: Range ['b' to 'd']

Entity Table

Entities Table

Entity Key	Serialized Entity
Foo:one	...
Foo:two	...
Bar:12	...
Bar:72	...

Kind:entity_id

These are nested for parent-child relationships:

ParentKind:parent_id/Kind:entity_id

Entities Table

```
class Grandparent(db.Model): pass
class Parent(db.Model): pass
class Child(db.Model): pass

ethel = Grandparent(key_name=Ethel)
jane = Parent(key_name=Jane, parent=ethel)
timmy = Child(key_name=Timmy, parent=jane)
```

```
Grandparent:Ethel
Grandparent:Ethel/Parent:Jane
Grandparent:Ethel/Parent:Jane/Child:Timmy
```

Entities Table

```
Grandparent:Alice
Grandparent:Alice/Parent:Sam
Grandparent:Ethel
Grandparent:Ethel/Parent:Jane
Grandparent:Ethel/Parent:Jane/Child:Timmy
Grandparent:Ethel/Parent:Jane/Child:William
Grandparent:Frank
```

Key name determines locality

Queries and Indexes

Kind index
Single-property index
Composite index
Merge join

Example GQL Queries

```
SELECT * FROM Person
  WHERE name = 'John';

SELECT * FROM Person
  ORDER BY name DESC;

SELECT * FROM Person
  WHERE city = 'Sonoma'
    AND state = 'CA'
    AND country = 'USA';

SELECT * FROM Person
  WHERE ANCESTOR IS :ethel
  ORDER BY name;
```

Indexes

- Separate Bigtable tables
- Map property values to entities
- Each row includes index data and entity key

Two kinds of tables

Entity Table

Key	Entity
a	...
b	...
c	...

Index Table

Index
foo
bar
baz

Entity Key is part of the key name in Index Tables, no columns

Scanning Indexes

- ~~Filtering in memory~~
- ~~Sorting in memory~~
- Not even a little!

Goal: convert query to dense index scan

- Pick the index
- Compute the prefix or range
- Scan!

Kind Index

Child	Grandparent:Ethel/Parent:Jane/Child:Timmy
Child	Grandparent:Ethel/Parent:Jane/Child:William
Grandparent	Grandparent:Alice
Grandparent	Grandparent:Ethel
Grandparent	Grandparent:Frank
Parent	Grandparent:Ethel/Parent:Jane
Parent	Grandparent:Alice/Parent:Sam

```
SELECT * FROM Grandparent
```

Single-property Index

Serves queries on a single property:

```
WHERE name = 'John'

ORDER BY name DESC

WHERE name >= 'B'
  AND name < 'C'
  ORDER BY name
```

One ascending, one descending

Single-Property Index

Kind	Prop Name	Prop Value	Key Name
Parent	address	1 Palm Dr.	...
Parent	name	Alice	...
Parent	name	Bob	...
Parent	name	Brad	...
Parent	name	Chelsea	...
Parent	name	Jane	...
Parent	name	John	...
Parent	title	Ninja Pirate	...

Text and Blob properties are not indexed

Descending Single-Property Index

Kind	Prop Name	Value (DESC)	Key Name
Parent	address	1 Palm Dr.	...
Parent	name	John	...
Parent	name	Jane	...
Parent	name	Chelsea	...
Parent	name	Brad	...
Parent	name	Bob	...
Parent	name	Alice	...
Parent	title	Ninja Pirate	...

Single Property Query

Kind	Prop Name	Prop Value	Key Name
Parent	address	1 Palm Dr.	...
Parent	name	Alice	...
Parent	name	Bob	...
Parent	name	Brad	...
Parent	name	Chelsea	...
Parent	name	Jane	...
Parent	**name**	**John**	**...**
Parent	title	Ninja Pirate	...

```
SELECT * FROM Parent WHERE name = 'John'

Index: Ascending
Prefix Scan: Parent name John
```

Single Property Query

Kind	Prop Name	Value (DESC)	Key Name
Parent	address	1 Palm Dr.	...
Parent	**name**	**John**	...
Parent	**name**	**Jane**	...
Parent	**name**	**Chelsea**	...
Parent	**name**	**Brad**	...
Parent	**name**	**Bob**	...
Parent	**name**	**Alice**	...
Parent	title	Ninja Pirate	...

```
SELECT * FROM Parent ORDER BY name DESC
```

```
Index: Descending
Prefix Scan: Parent name
```

Single Property Query

Kind	Prop Name	Value (DESC)	Key Name
Parent	address	1 Palm Dr.	...
Parent	name	Alice	...
Parent	**name**	**Bob**	...
Parent	**name**	**Brad**	...
Parent	name	Chelsea	...
Parent	name	Jane	...
Parent	name	John	...
Parent	title	Ninja Pirate	...

```
SELECT * FROM Parent
  WHERE name >= 'B' AND name < 'C'
  ORDER BY name
```

Index: Ascending

Range Scan: [Parent name B, Parent name C)

Composite Index

- Defined in index.yaml
- Multiple properties or ancestor

```
WHERE firstname = 'John'
  AND lastname = 'Smith'

WHERE firstname >= 'B' AND firstname
 < 'C' AND lastname = 'Smith'

WHERE ANCESTOR IS :ethel
  ORDER BY firstname
```

index.yaml

```yaml
- kind: Parent
  properties:
  - name: lastname
  - name: firstname
```

index.yaml

```yaml
- kind: Parent
  properties:
  - name: lastname
  - name: firstname
```

Kind	lastname	firstname	Key Name
Parent	Anderson	Jane	...
Parent	Barrett	Ryan	...
Parent	Smith	Bob	...
Parent	Thomas	Alice	...

Query Planning

1. Add kind to prefix
2. Add ancestor, if any
3. Add = filters, if any
4. If any inequality filters, convert to range scan
5. Add sort orders, if any

Composite Index

Kind	lastname	firstname	Key Name
Parent	Anderson	Jane	...
Parent	Barrett	Ryan	...
Parent	**Smith**	**Bob**	**...**
Parent	Thomas	Alice	...

```
SELECT * from Parent
  WHERE firstname = 'Bob'
  AND lastname = 'Smith'
```

Prefix: Parent Smith Bob:

Composite Index

Kind	lastname	firstname	Key Name
Parent	Anderson	Jane	...
Parent	Barrett	Ryan	...
Parent	**Smith**	**Bob**	**...**
Parent	**Smith**	**Brad**	**...**
Parent	Smith	Alice	...

```
SELECT * FROM Parent
  WHERE lastname = 'Smith'
  AND firstname >= 'B'
  AND firstname < 'C'
```

Range Scan:

```
[Parent Smith B, Parent Smith C)
```

Ancestor Queries

```
- kind: Parent
  ancestor: yes
  properties:
  - name: firstname
```

Kind	Ancestor	firstname	Key Name
Parent	/Grandparent:David	Jane	...
Parent	/Grandparent:Ethel	Alice	...
Parent	/Grandparent:Ethel	Bob	...
Parent	/Grandparent:Ethel	Ryan	...

Ancestor Queries

Kind	Ancestor	firstname	Key Name
Parent	/Grandparent:David	Jane	...
Parent	/Grandparent:Ethel	Alice	...
Parent	/Grandparent:Ethel	Bob	...
Parent	/Grandparent:Ethel	Ryan	...

```
SELECT * FROM Parent
  WHERE ANCESTOR IS /Grandparent:Ethel
  ORDER BY firstname
```

Prefix Scan: Parent /Grandparent:Ethel

Merge Join

Serves multiple '=' filters without a composite index

Can include ancestor

```
WHERE firstname = 'John'
  AND lastname = 'Smith'
```

Scan single-property index once for each filter value and intersect results

Merge Join

Kind	Prop Name	Prop Value	Key Name
Parent	firstname	Alice	X
Parent	firstname	Bob	Y
Parent	**firstname**	**John**	**Z**

Prefix: Parent firstname John

Kind	Prop Name	Prop Value	Key Name
Parent	lastname	Barrett	D
Parent	lastname	Smith	E
Parent	**lastname**	**Smith**	**Z**

Prefix: Parent lastname Smith

```
WHERE firstname = 'John' AND lastname = 'Smith'
```

Transactions

Transactions and Entity Groups

Transaction model

All writes are transactional

Timestamped journals

Entity version history

Optimistic concurrency

Example: PUT

Read Committed timestamp

	Key	Entity	Journal	Committed
←	A	X (3:00)	...	(3:00)

Example: PUT

Read Committed timestamp

Write Journal

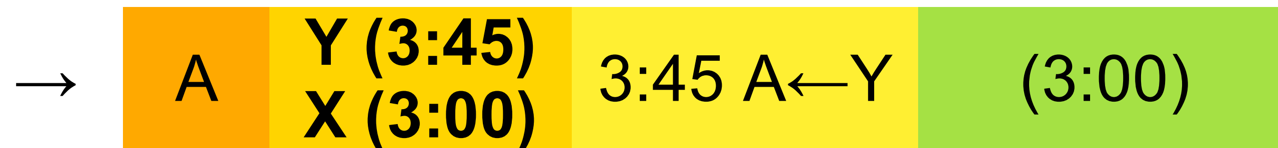
	Key	**Entity**	**Journal**	**Committed**
→	A	X (3:00)	**3:45 A←Y**	(3:00)

Example: PUT

Read Committed timestamp

Write Journal

Apply Journal



Example: PUT

Read Committed timestamp

Write Journal

Apply Journal

Update Committed timestamp in a Bigtable row transaction

← →	A	Y (3:45) X (3:00)	3:45 A←Y	~~(3:00)~~ **(3:45)**

Committed is only updated if old value of Committed is consistent.

Entity Groups

Defined by root entities

Consists of root and all descendants

Journal, last committed timestamp
co-located with root entity

Entities Table

Entity Key	Serialized Entity
Foo:one	...
Foo:two	...
Bar:12	...
Bar:72	...

Kind:entity_id

These are nested for parent-child relationships:

ParentKind:parent_id/Kind:entity_id

Example: Paying Allowance

```
class Parent(db.Model):
    cash = db.IntegerProperty()

class Child(db.Model):
    cash = db.IntegerProperty()

parent = Parent(cash=1000)
child = Child(parent=parent, cash=0)
```

Example: Paying Allowance

```
def pay(parent_key, child_key, amount):
  parent, child = db.get(parent_key, child_key)
  parent.cash -= amount
  child.cash += amount
  db.put(parent, child)

db.run_in_transaction(
  pay, parent.key(),child.key(), 10
)
```

Example: Paying Allowance

```
def pay(parent_key, child_key, amount):
   parent, child = db.get(parent_key, child_key)
   parent.cash -= amount
   child.cash += amount
   db.put(parent, child)

db.run_in_transaction(
   pay, parent.key(),child.key(), 10)
```

Read entity group for parent
Note committed timestamp, 3:00pm
Read parent and child entities as of 3:00pm

Example: Paying Allowance

```
def pay(parent_key, child_key, amount):
   parent, child = db.get(parent_key, child_key)
   parent.cash -= amount
   child.cash += amount
   db.put(parent, child)

db.run_in_transaction(
   pay, parent.key(),child.key(), 10)
```

Serialize parent and child
Write to journal in parent's entity group row

Example: Paying Allowance

```
def pay(parent_key, child_key, amount):
   parent, child = db.get(parent_key, child_key)
   parent.cash -= amount
   child.cash += amount
   db.put(parent, child)

db.run_in_transaction(
   pay, parent.key(),child.key(), 10)
```

Apply journalled changes to parent and child
Write new committed timestamp to entity group
Update parent.cash and child.cash index rows

Recovery

Always check entity group first

Unfinished transactions are rolled forward

Adds one Bigtable read to every
datastore read/write

Questions

http://tinyurl.com/ete-hood

Thank You