Announcements

- Test 1 feedback
- GCP billing errors
Why non-relational systems?

• Need for greater scalability
  • Throughput
  • Response time

• More expressive data models and schema flexibility

• Object-relational mismatch

• Preference for open-source software
Why Firestore?

- Document database system
- Fully serverless
- Integrated with GCP
- Simple APIs for reading and writing
- Supports transactions
- Provides strong consistency (uses Spanner for storage)
- Designed for mobile, web and IoT apps
- Comes in two modes: native and datastore
- Clients can listen for document updates (native mode only)
- Massive scale (10+M requests/sec, PBs of storage)
- Write throughput limits in native mode (10K writes/sec)
Firestore’s Data Model

- Firestore is a document database system
- Firestore document == set of typed key, value pairs
- Primitive types: String, Int, Float, Bool, Datetime
- Complex types: Array, Map, Geo points

- Documents are grouped into collections
- Documents of the same type can have different schemas
- Documents have unique identifiers (id)
- Documents can store hierarchical data with subcollections
Writing to Firestore

- Set method converts Python dictionary into Firestore document
- Every document has unique identifier
- Writes must also update indexes on documents

Example 1: writes into author collection
```
from google.cloud import firestore
db = firestore.Client()

author = {
    'id': 'aaa',
    'name': 'Mary Tuma',
    'section': 'news',
    'active': True,
    'start_date': '2019-01-20'
}

db.collection('author').document('aaa').set(author)
```

Example 2: writes into article collection
```
from google.cloud import firestore
db = firestore.Client()

article = {
    'id': '1',
    'title': 'Turmoil at the Zoo',
    'published': True,
    'publication_date': '2019-01-26',
    'auth_id': 'aaa',
    'clicks': 120,
    'likes': 45,
    'dislikes': 9,
    'comments': 13
}

db.collection('article').document('1').set(article)
```
Writing to Firestore

Example 3: writes into tag collection

```javascript
const tag = {
  'id': '1',
  'tag': 'politics',
  'article_ids': ['1', '2', '3', '4', '5', '6', '7']
}

db.collection('tag').document('1').set(tag)

tag = {
  'id': '2',
  'tag': 'austin',
  'article_ids': ['1', '8', '9', '10']
}

db.collection('tag').document('2').set(tag)
```

Example 4: writes into nested_article collection

```javascript
tag1 = {
  'id': '1',
  'tag': 'politics',
}
tag2 = {
  'id': '2',
  'tag': 'news',
}
tags = []
tags.append(tag1)
tags.append(tag2)
nested_article = {
  'id': '1',
  'title': 'Turmoil at the Zoo',
  'published': True,
  'publication_date': '2019-01-26',
  'auth_id': 'aaa',
  'clicks': 120,
  'likes': 45,
  'dislikes': 9,
  'comments': 13,
  'tags': tags
}

db.collection('nested_article').document('1').set(nested_article)
```
Reading from Firestore

- Get(id) method fetches single document
- Stream method fetches all documents in collection
- Stream + where methods filter documents in collection
- Order by and limit methods available
- All reads require indexes!

Example 1: reads single document
```python
doc_ref = db.collection('author').document('aaa')
doc = doc_ref.get()
if doc.exists:
    print(f'{doc.id} => {doc.to_dict()}')
else:
    print('No such author!')
```

Example 2: reads all documents in collection
```python
docs = db.collection('article').stream()
for doc in docs:
    print(f'{doc.id} => {doc.to_dict()}')
```

Example 3: filters documents in collection
```python
docs = db.collection('author').where('name', '==', 'Nina Hernandez').stream()
for doc in docs:
    print(f'{doc.id} => {doc.to_dict()}')
```
Document Database Design Principles

1. Know problem domain and understand usage patterns.
2. Group entities into top-level and lower-level types.
3. Make each top-level entity type its own Firestore collection.
4. Embed lower-level entities into their related top-level entity when they share a 1:m relationship.
5. Merge lower-level entities with their related top-level entity when they share a 1:1 relationship.
6. Eliminate m:n relationships by embedding both sides of the relationship into parent entities.
Schema conversion example

College Normalized Database

```
Teacher
PK  tid  VARCHAR
    fname  VARCHAR
    lname  VARCHAR
    dept   VARCHAR

Teaches
PK, FK  tid  VARCHAR
        cno  VARCHAR

Student
PK  sid  VARCHAR
    fname  VARCHAR
    lname  VARCHAR
    dob   DATE
    status CHAR

Takes
PK, FK  sid  VARCHAR
        cno  CHAR
        grade  CHAR

Class
PK  cno   CHAR
    cname  VARCHAR
    credits INT
```
Schema conversion example

**College Denormalized Database**

**Teacher**
- id
- tid
- fname
- lname
- dept
- cno
  - ARRAY<STRING>

**Student**
- id
- sid
- fname
- lname
- dob
- status
- classes
  - ARRAY<MAP>

**Class**
- id
- cno
- cname
- credits
  - STRING
  - INT

**Legend**
- Collections in green
- Embedded maps in yellow
Practice Problem 1

How would you remodel the Shopify database for Firestore?
Set up Firestore

Practice Problem 2

Find all classes taught by Prof. Cannata. Return their cid.
Project 4