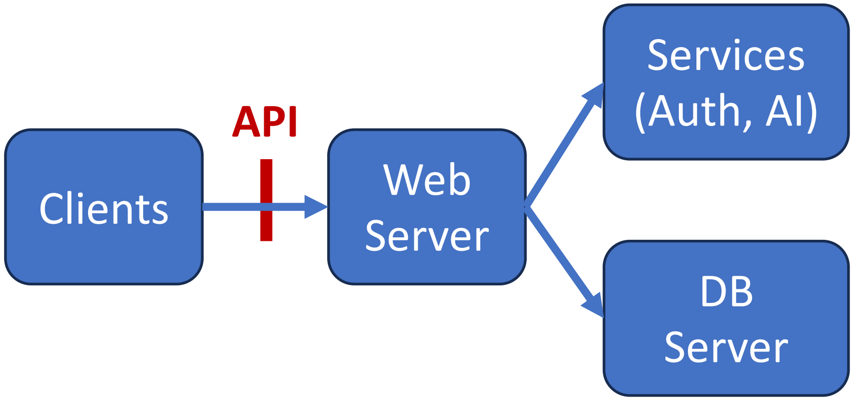
# Firestore Queries & Cloud Functions

Shan-Hung Wu

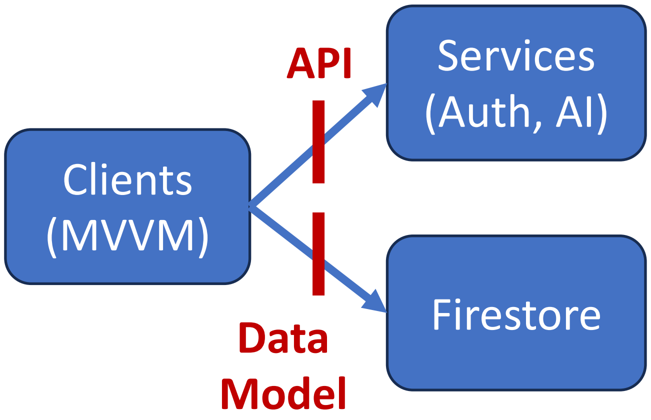
CS, NTHU

# Firestore as Backend Database

## Traditional App Architecture



## Architecture w. Firestore



- Data model exposed, not just APIs
- ViewModels transform data into state for Views
- ***Security rules*** required
  - E.g., each user should only be able to modify her own to-do items
  - Needs authentication; to be discussed later

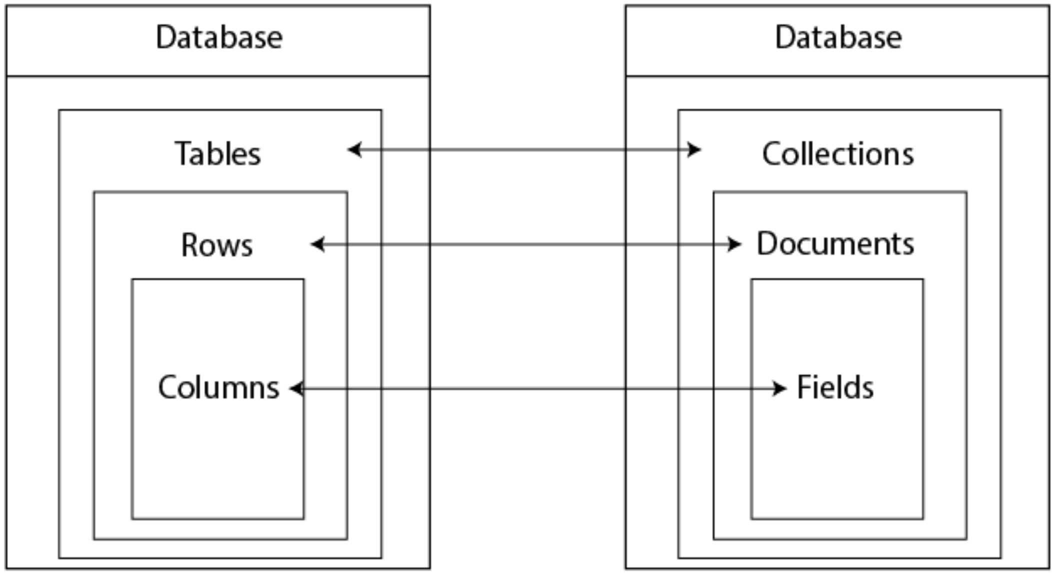
# Mastering Firestore

- SQL vs. NoSQL database Systems
- Queries
  - `where`, `sort`, and indexes
  - Maps and arrays
  - Pagination
  - To listen or get?
- Offline support
- Cloud Functions

# SQL vs. NoSQL DB Systems

RDBMS

NoSQL



Database

Tables

Rows

Columns

Database

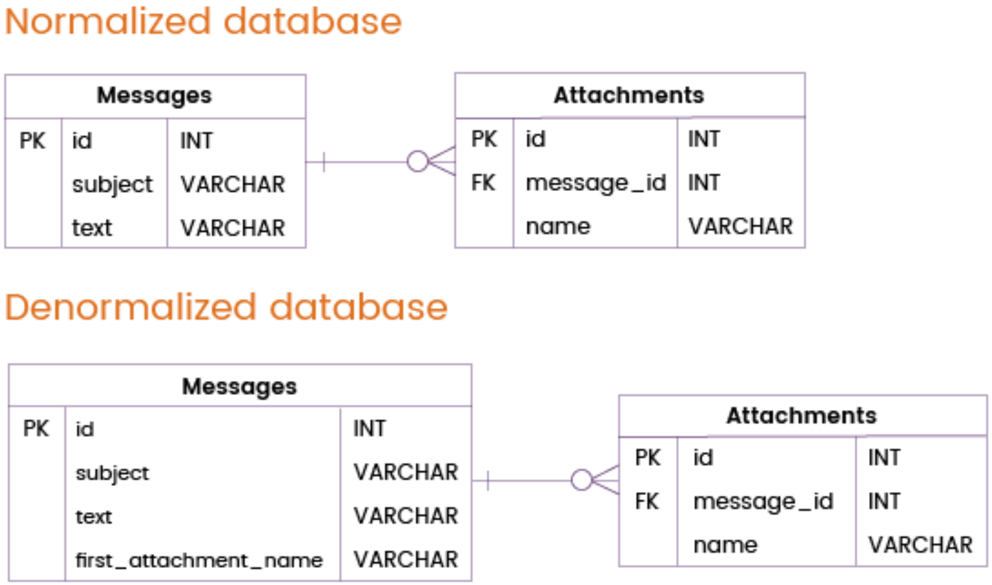
Collections

Documents

Fields

- Relational DB systems (RDBMS): MySQL, AWS RDS…
- NoSQL DB systems: MongoDB, Firestore…

# Features of RDBMS



Normalized database

| Messages | | |
|---|---|---|
| PK | id | INT |
| | subject | VARCHAR |
| | text | VARCHAR |

| Attachments | | |
|---|---|---|
| PK | id | INT |
| FK | message_id | INT |
| | name | VARCHAR |

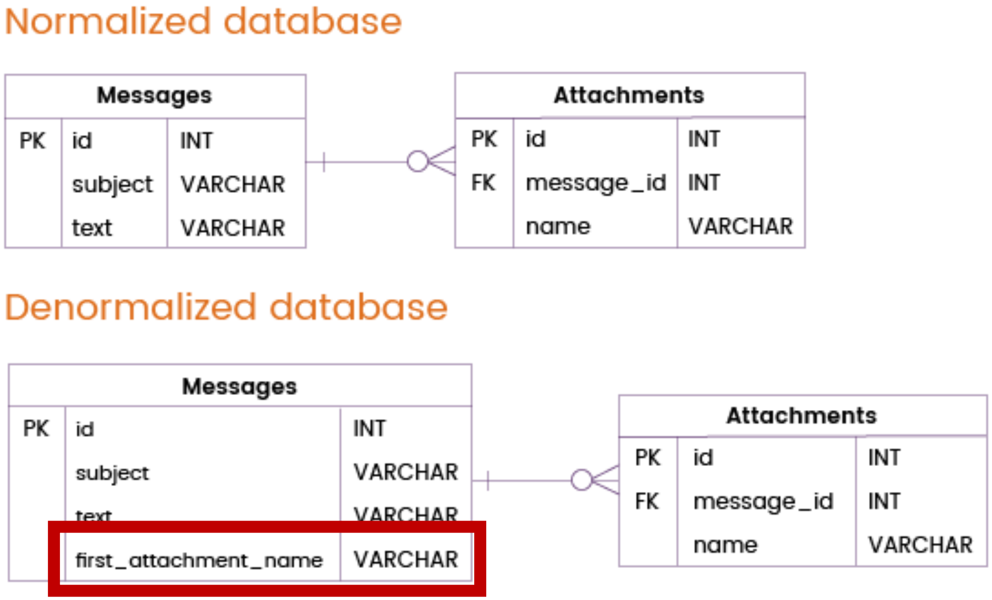
Denormalized database

| Messages | | |
|---|---|---|
| PK | id | INT |
| | subject | VARCHAR |
| | text | VARCHAR |
| | first_attachment_name | VARCHAR |

| Attachments | | |
|---|---|---|
| PK | id | INT |
| FK | message_id | INT |
| | name | VARCHAR |

- Strick data models (RE and relational models) avoiding duplicated data
- Supports complex (join) queries in SQL language
- Conservative concurrency control (via locking protocols)
- Scale up (on high-end machines)

# Features of NoSQL DB Systems



Normalized database

| Messages | | |
|---|---|---|
| PK | id | INT |
| | subject | VARCHAR |
| | text | VARCHAR |

| Attachments | | |
|---|---|---|
| PK | id | INT |
| FK | message_id | INT |
| | name | VARCHAR |

Denormalized database

| Messages | | |
|---|---|---|
| PK | id | INT |
| | subject | VARCHAR |
| | text | VARCHAR |
| | first_attachment_name | VARCHAR |

| Attachments | | |
|---|---|---|
| PK | id | INT |
| FK | message_id | INT |
| | name | VARCHAR |

- Embrace data ***duplication***/***de-normalization***
- Limited query capabilities, but with ***listening*** & ***offline support***
- Optimistic concurrency control (OCC) through auto-retries
    - To cope with lost clients
- Scale out (across many commodity machines)

# Mastering Firestore

- SQL vs. NoSQL database Systems
- Queries
  - `where`, `sort`, and indexes
  - Maps and arrays
  - Pagination
  - To listen or get?
- Offline support
- Cloud Functions

# Queries

```
QuerySnapshot querySnapshot = await _db
    .collection('employees')
    .where('age', isEqualTo: 25) // predicate
    .get();

for (QueryDocumentSnapshot doc in querySnapshot.docs) {
  Map<String, dynamic> data = doc.data();
  print('Age: ${data['age']}');
}
```

- You can only query docs ***in same collection***
- Exception: in multiple collections with same name via collection-group queries

# Indexes

```
QuerySnapshot querySnapshot = await _db
    .collection('employees')
    .where('age', isEqualTo: 25)
    .get();
```

- Sorted lists of (field values → doc ID) pairs

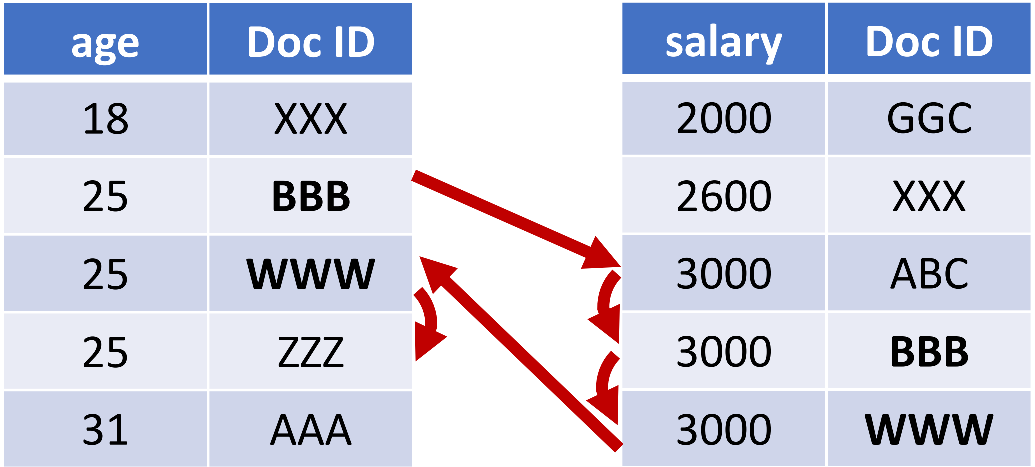
| age | Doc ID |
|---|---|
| 18 | XXX |
| 25 | BBB |
| 25 | WWW |
| 25 | ZZZ |
| 31 | AAA |

- Enable binary searches
- Single-field indexes created automatically
  - [Two indexes (ASC and DEC) for each field]

# Multiple Equality Constraints

```
QuerySnapshot querySnapshot = await _db
    .collection('employees')
    .where('age', isEqualTo: 25)
    .where('salary', isEqualTo: 3000)
    .get();
```

- Need ***no*** extra indexes
- Doc IDs are secondary sorted
- Can use Zig-zag merge join



| age | Doc ID |
|---|---|
| 18 | XXX |
| 25 | **BBB** |
| 25 | **WWW** |
| 25 | ZZZ |
| 31 | AAA |

| salary | Doc ID |
|---|---|
| 2000 | GGC |
| 2600 | XXX |
| 3000 | ABC |
| 3000 | **BBB** |
| 3000 | **WWW** |

# Single Inequality Constraints

```
QuerySnapshot querySnapshot = await _db
    .collection('employees')
    .where('age', isEqualTo: 25)
    .where('salary', isGreaterThanOrEqualTo: 3000)
    .orderBy(salary', descending: true)
    .get();
```

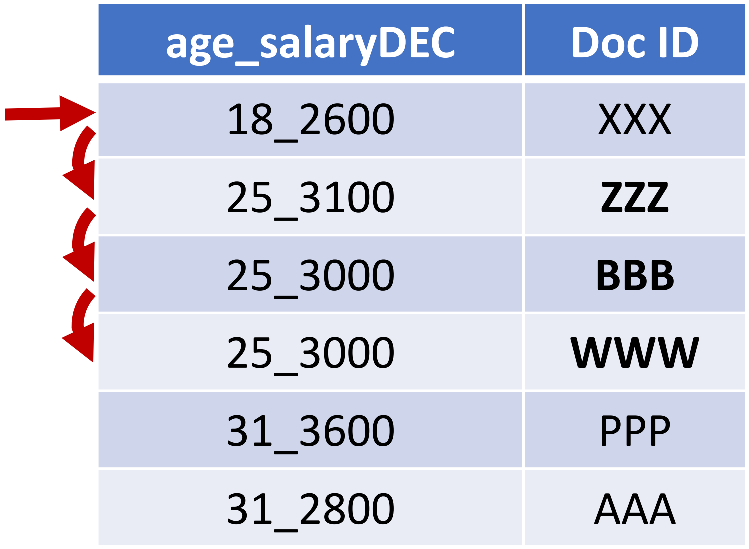
- No zig-zag; ***composite indexes*** needed
- ***Not*** created automatically
  - Too many: O($2^n$) for *n* fields
  - Follow "query requires an index" error message to create one
- <500 per project

| age_salaryDEC | Doc ID |
|---|---|
| 18_2600 | XXX |
| 25_3100 | ZZZ |
| 25_3000 | BBB |
| 25_3000 | WWW |
| 31_3600 | PPP |
| 31_2800 | AAA |

# Multiple Inequality Constraints

```
QuerySnapshot querySnapshot = await _db
    .collection('employees')
    .where('age', isLessThanOrEqualTo : 25)
    .where('salary', isGreaterThanOrEqualTo: 3000)
    .orderBy(age')
    .orderBy(salary', descending: true)
    .get();
```

- Composite indexes needed
- Use index scan that reads entries ***not*** in query results
  - Costs: 1000 index entry reads = 1 doc read

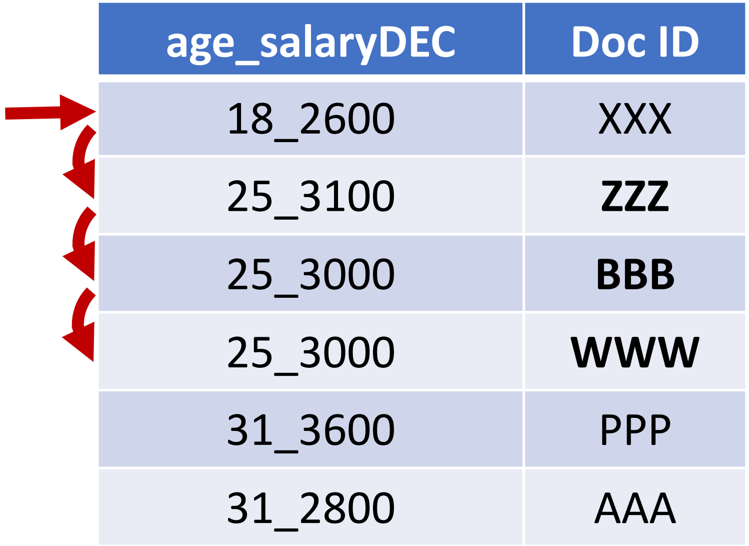


| age_salaryDEC | Doc ID |
|---|---|
| 18_2600 | XXX |
| 25_3100 | **ZZZ** |
| 25_3000 | **BBB** |
| 25_3000 | **WWW** |
| 31_3600 | PPP |
| 31_2800 | AAA |

# Ordering Fields in Composite Index

```
QuerySnapshot querySnapshot = await _db
    .collection('employees')
    .where('age', isLessThanOrEqualTo : 25)
    .where('salary', isGreaterThanOrEqualTo: 3000)
    .orderBy(age')
    .orderBy(salary', descending: true)
 .get();
```

- The first field in a composite index matters
  - 10000 total docs
  - 50% matched age & 1% matched salary
  - → ***5000*** index entry reads + 50 doc reads

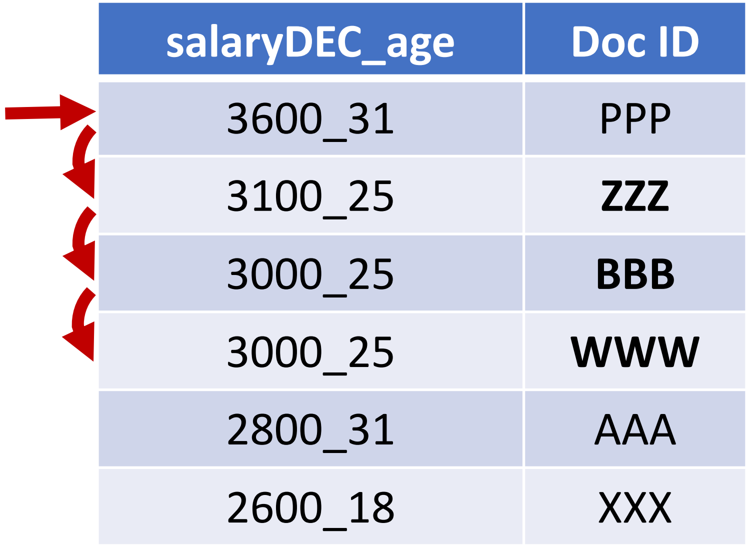


| age_salaryDEC | Doc ID |
|---|---|
| 18_2600 | XXX |
| 25_3100 | **ZZZ** |
| 25_3000 | **BBB** |
| 25_3000 | **WWW** |
| 31_3600 | PPP |
| 31_2800 | AAA |

# Better Query

```
QuerySnapshot querySnapshot = await _db
     .collection('employees')
     .where('age', isLessThanOrEqualTo : 25)
     .where('salary', isGreaterThanOrEqualTo: 3000)
     .orderBy(salary', descending: true)
     .orderBy(age')
     .get();
```

- Order fields in decreasing order of query constraint selectivity
  - 10000 total docs
  - 50% matched age & 1% matched salary
  - → *100* index entry reads + 50 doc reads



| salaryDEC_age | Doc ID |
|---|---|
| 3600_31 | PPP |
| 3100_25 | **ZZZ** |
| 3000_25 | **BBB** |
| 3000_25 | **WWW** |
| 2800_31 | AAA |
| 2600_18 | XXX |

# Mastering Firestore

- SQL vs. NoSQL database Systems
- Queries
  - `where`, `sort`, and indexes
  - Maps and arrays
  - Pagination
  - To listen or get?
- Offline support
- Cloud Functions

# Maps and Arrays

- A doc field can be a map or array
- Each field in a map is also indexed automatically

| address.city | Doc ID |
|---|---|
| “Taipei” | … |
| “Hsin Chu” | … |
| “New York” | … |

- Why subcollections then?
  - Doc size <1 MB, #fields < 20K
  - 1 write per second for same doc
  - Subcollection can be partially retrieved (via queries)

```
Employee { // doc
  name: ...,
  address: {
    city: ...,
    street1: ...,
    street2: ...,
    ...
  },
  languages: [
    'C++',
    'Dart',
    ...
  ],
  ...
}
```

# Queries on Maps

- Find users in city “Taipei”:

```
_db.collection('employees')
   .where(
     address.city,
     isEqualTo : 'Taipei',
   ).get();
```

- Find users with 2 street lines?

```
_db.collection('employees')
   .where(
     address.street2,
     isGreaterThandOrEqualTo : '',
   ).get();
```

```
Employee { // doc
  name: ...,
  address: {
    city: ...,
    street1: ...,
    street2: ...,
    ...
  },
  languages: [
    'C++',
    'Dart',
    ...
  ],
  ...
}
```

# Queries on Arrays

```
Employee { // doc
  name: ...,
  address: {
    city: ...,
    street1: ...,
    street2: ...,
    ...
  },
  languages: [
    'C++',
    'Dart',
    ...
  ],
  ...
}
```

- To avoid concurrency problems, no access to element's index
  - No `devices[i]`
  - No `insertAt()` / `updateAt()`

- Think of array as "a set of flags":

```
_db.collection('employees')
   .where(
     languages,
     arrayContains : 'Dart',
     // or arrayContainsAny: ['Dart', 'Java'],
   ).get();
```

# Array Indexes

- Firestore treats arrays as maps
  - Uses binary search on secondary-sorted Doc IDs

| language.Dart | Doc ID |
|---|---|
| true | … |
| true | PPP |
| true | … |
| … | … |

```
// doc field
languages: [
  'C++',
  'Dart',
  ...
]

// query
arrayContains: 'Dart'
```

→

```
// doc field
languages: {
  'C++': true,
  'Dart': true,
  ...
}

// query
languages.Dart = true
```

# Mastering Firestore

- SQL vs. NoSQL database Systems
- Queries
  - `where`, `sort`, and indexes
  - Maps and arrays
  - Pagination
  - To listen or get?
- Offline support
- Cloud Functions

# Pagination (1/2)

- In repository:

```
DocumentSnapshot? _lastDoc;

Future<List<Employee>> getPage(bool isFirst){
  Query query = await _db
      .collection('employees')
      .orderBy('age')
      .limit(20);
  // _lastDoc's field values are used to locate the start position
  // in the index
  if (!isFirst && _lastDoc != null)
    query = query.startAfterDocument(_lastDoc!);

  QuerySnapshot snapshot = await query.get();

  if (snapshot.docs.isNotEmpty) _lastDoc = snapshot.docs.last;
  return snapshot.docs.map((doc) => ...).toList();
}
```

**Infinite Scroll**

# Pagination (2/2)

- In view:

```
ListView.builder(
  itemCount: _employees.length,
  itemBuilder: (context, index) {
    // pre-fetch page
    if (index >= _employees.length - 5) {
      List<Employee> page =
          await _repository.getPage(false);
      if (page.isNotEmpty) {
        setState(() {
          _employees.addAll(page);
        });
      };
    }
    return ListTile(_employees[index]);
  },
),
```

Infinite Scroll

# Mastering Firestore

- SQL vs. NoSQL database Systems
- Queries
  - `where`, `sort`, and indexes
  - Maps and arrays
  - Pagination
  - To listen or get?
- Offline support
- Cloud Functions

# Should I Listen to Query Results?

- Generally yes, except:
- ***Pagination***
  - Update of a single doc may affect all pages
  - Listen to (inconsistent) last page or all pages (cost)?
- Results change more often than ***user expectation***
  - Group chat, group notes, multiplayer games ✅
  - Stock market prices, leaderboards ✅
  - Social feed ❌
  - Statistics ❌
  - User profile and avatar ❌
- You don’t want to pay the ***costs***

# Mastering Firestore

- SQL vs. NoSQL database Systems
- Queries
  - `where`, `sort`, and indexes
  - Maps and arrays
  - Pagination
  - To listen or get?
- Offline support
- Cloud Functions

# Types of Being “Offline”

- ***Disconnected***: no physical connections
  - Cellular OFF
  - Wi-Fi OFF, etc.
- ***Isolated***: connected, but no route to Internet
  - Low-quality connections
  - Authentication required
  - Firewall restrictions
  - VPN Issues, etc.

- Firestore deisgned for “occasional” offline scenarios

# Persistent Caching

- Enabled in mobile SDKs by default
  - ***Not*** in web SDK due to issues like shared browser, compatibility, multi-tabs, etc.
  - To enable:

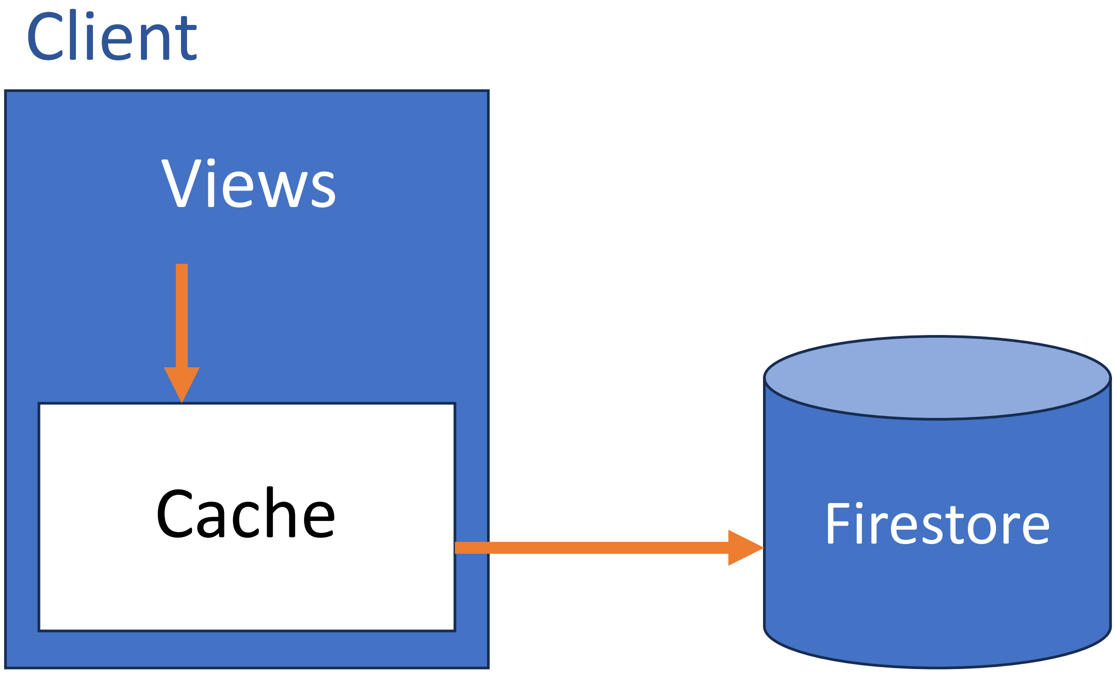
```
// may return error!
await _db.enablePersistence(
    const PersistenceSettings(
        synchronizeTabs: true));
```

- Size configurable:

```
db.settings = const Settings(
  persistenceEnabled: true,
  cacheSizeBytes: Settings.CACHE_SIZE_UNLIMITED,
);
```

  - Lease recently used data are replaced when full

# Updates



- Per doc:
  - ***To local cache first***
  - Then on server when client goes back online
- Conflict resolution (on same doc): the last write wins
  - Earlier offline update could win over later online updates
- Transactions ***fails*** (except batch writes)
  - Catch errors or disable corresponding UI in advance

# Post-Update Code in UI

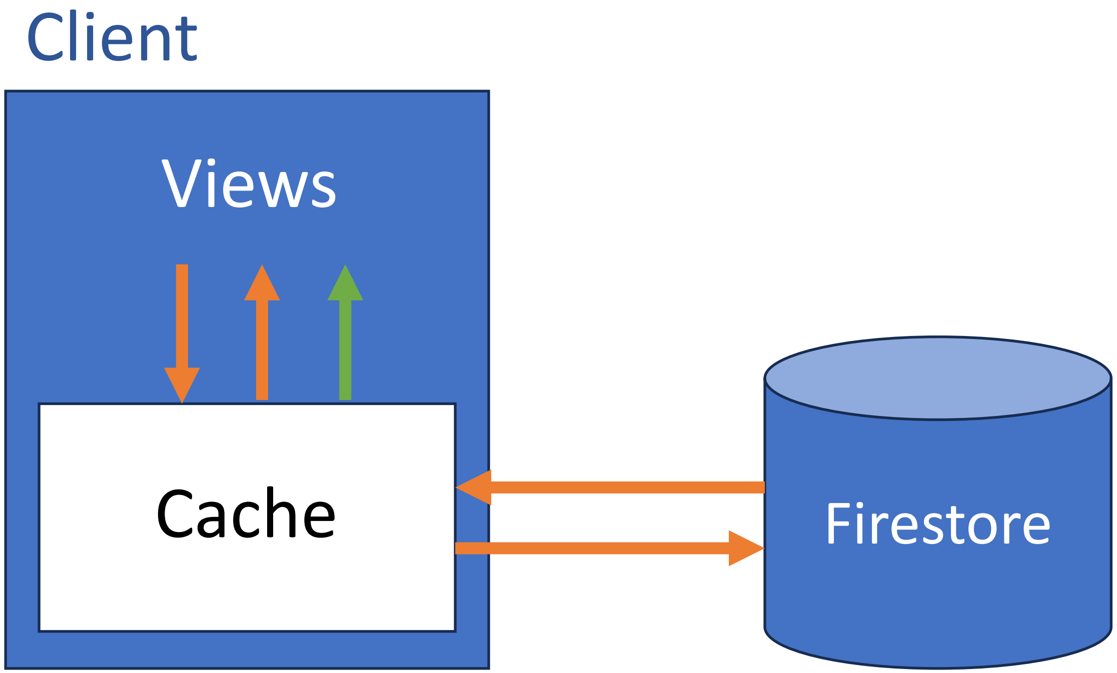
- Closing a dialog after updating a doc like this?

```
await _db.collection('employees').add(...)
... // close dialog
```

- UI hangs!
  - Future resolved only after server updates the doc
- Since Firestore writes to local cache immediately, simply write your code as:

```
_db.collection('employees').add(...)
... // close dialog
```

# Listening



- Listeners may be notified twice
  - ***Local copy first***, then server copy (if data change)
- Users always see the changes immediately
- Same flow for disconnected and isolated cases

# Distinguishing Local from Server Events

```
db.collection('employees')
  .where('salary', isEqualTo: 300)
  .snapshots(includeMetadataChanges: true)
  .listen((querySnapshot) {
    if (querySnapshot.metadata.isFromCache) {
      ...
    }
  });
```

- Optionally, set `includeMetadataChanges` to true if you always want listeners to be notified twice
  - Useful for, e.g., showing “Syncing…” satus in UI

# Gets

- Disconnected: return data from cache
- Isolated: return data from cache after timeout
  - Possible improvement: your own cache strategy

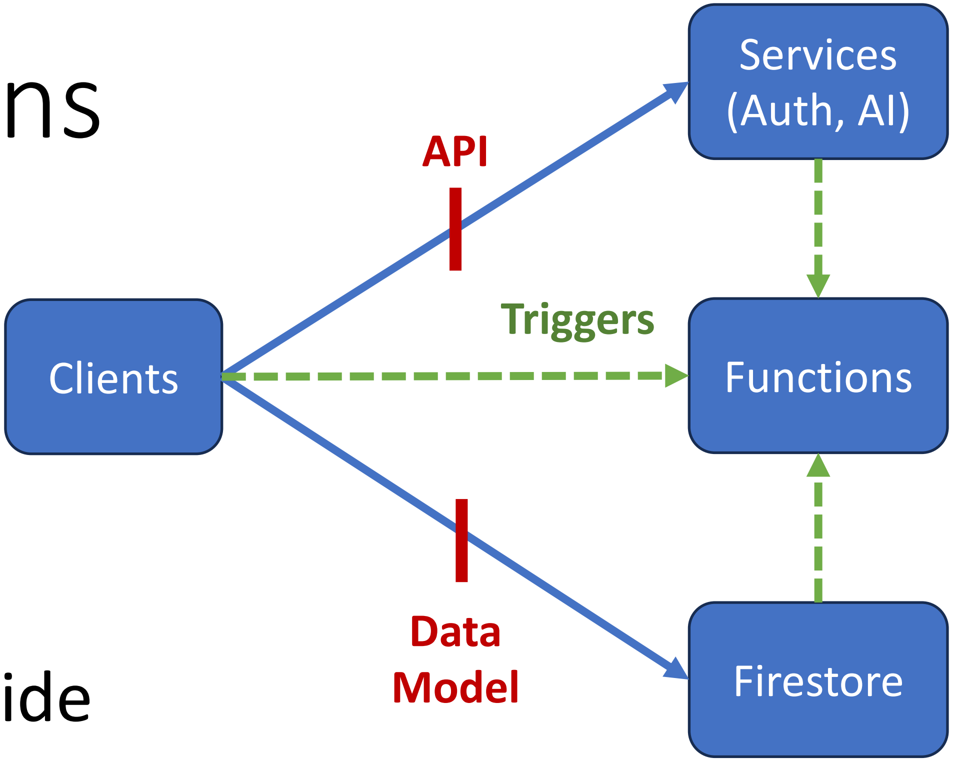
```
// If same query is issued again within 15 min
_db.collection('employees')
  .where(...)
  .get(GetOptions(source: Source.cache))

... // Recall get() if data is updated locally
```

- Offline gets can be new queries
  - Executed locally against local data
- What if there’s no cached data?
  - Collection: empty collection returned
  - Doc: error!

# Mastering Firestore

- SQL vs. NoSQL database Systems
- Queries
  - `where`, `sort`, and indexes
  - Maps and arrays
  - Pagination
  - To listen or get?
- Offline support
- Cloud Functions

# Cloud Functions



- Executed at server side
- V1 & V2
  - V2 is faster and more scalable, but does not support authentication and analytics triggers currently
- Supported languages: ***Javascript*** or Python
  - The "/functions" folder is a Node.js project

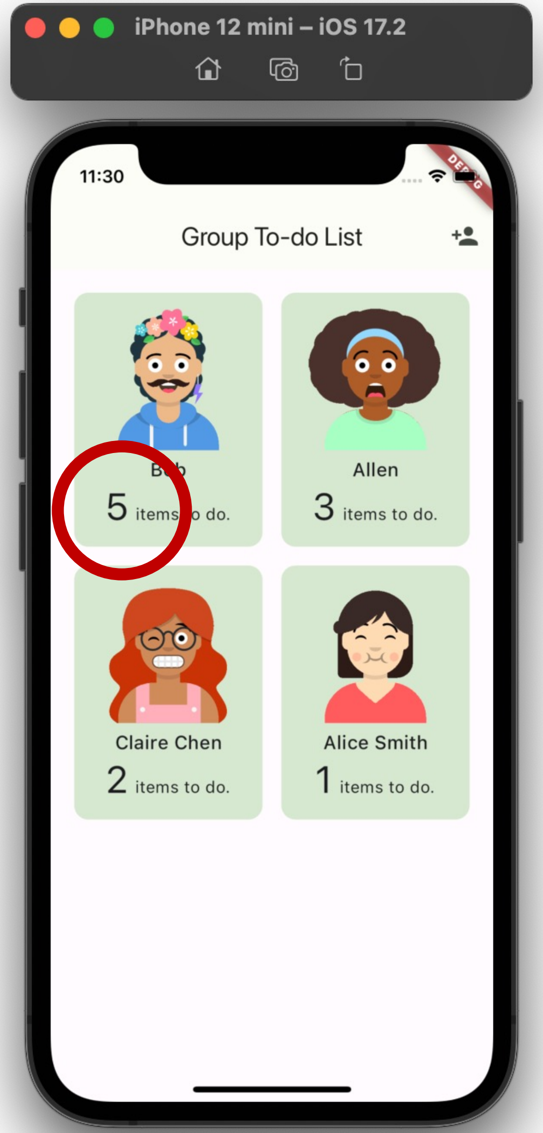
# Usage

- Detect Firestore changes and run post-change logic
- Send push notifications
- Save images to Cloud Storage
- Call 3rd-party services (e.g., OpenAI APIs)
- Handle HTTP requests
- Execute cron jobs periodically
- Talk to pub/sub channels
- …
- These are “background” tasks with ***delays***

# Syncing Denormalized Data

- Done in Functions to pass security rules (if any)
  - "Each user should only be able to modify her own to-do items"
- How?

1. Detect to-do item creation / deletion
2. Run a transaction to
   - Increase/decrease `User.itemCount`
   - Record processed time to ensure idempotency



# Idempotency

- In a large data center, errors are norm rather than exceptions
- An event (e.g., doc creation or deletion) with same ID may be triggered more than once
- Each of your functions needs to be ***idempotent***
  - Multiple calls = single call
- How?
  - Use event IDs as idempotency keys
  - Record processing time for each key in a transaction
  - Skip processing if key already exists

# Transactions in Cloud Functions

- ***Can run queries*** in the “read” part
  - Different from client-side transactions, which only allow reading individual docs
- Pessimistic concurrency control based on locking protocol
  - Different from client-side transactions, which uses optimistic concurrency control (OCC)
- Limitations:
  - <10 MB reads
  - <500 writes

# Remarks

- Cloud Functions bypass security rules
  - Server code is written by you and can be trusted
- Each function runs in separated container
  - Warm-up delay
  - Global variables are actually local to container
- Lazily load a heavy-weight variable/package inside the function that needs it
- Functions may not run in order of events
  - Event order: user sign up → user doc created
  - Functions for the two events may be out of order

# Further Readings

- More about Firestore:
- Aggregations queries
  - E.g., count, sum, average, etc.
- Vector searches

- More about Cloud Functions:
- Handling HTTP requests
- Schedule functions