Data Model of Megastore

Learn about the data model of Megastore.

We'll cover the following

This lesson discusses the trade-offs between traditional databases and NoSQL and describes the resulting developer-facing features in Megastore's development.

API design

In traditional databases, to facilitate user operations, normalized relational schemas depend on joins at query time. This is not the proper approach for Megastore applications because of the following reasons:

- Relatively stable performance benefits high-volume interactive workloads much more than an expressive query language, for example, SQL provides many types of joins.
- It is beneficial to shift work from read time to write time since there are more reads than writes in the applications where Megastore is to be used. Traditional database queries are translated into an execution plan, and then this plan is optimized when a query is submitted.
- In key-value systems like Bigtable, storing and accessing hierarchical data is simple. Therefore, we might not need extensive query optimization.

Considering these factors, we have created a data model and schema language that gives users granular control over where their data is stored.

Data model

Megastore proposes a data model that sits somewhere between an RDBMS's abstract tuples and NoSQL's actual row-column storage. The data model is defined in a schema and strongly typed, similar to RDBMS. Every schema consists of a collection of tables, each of which has a collection of entities that in turn have a collection of attributes. Property values are named and typed. Google's Protocol Buffers, strings, and various numeric representations are all acceptable types. We can define them as optional, required, or repeated (A single property can have a number of values.) The table's entities all have the same set of permitted attributes. The entity's primary key is assembled by a series of properties, and the primary keys must be distinctive inside the table. Let's see an example of the database structure for a basic picture album app in the code below:

```
 1  CREATE SCHEMA PhotoApp;
 2
 3
 4  CREATE TABLE User {
 5    required int64 user_id;
 6    required string name;
 7  } PRIMARY KEY(user_id), ENTITY GROUP ROOT;
 8
 9
10  CREATE TABLE Photo {
11    required int64 user_id;
12    required int32 photo_id;
13    required int64 time;
14    required string full_url;
15    optional string thumbnail_url;
16    repeated string tag;
17  } PRIMARY KEY(user_id, photo_id), IN TABLE User,
18  ENTITY GROUP KEY(user_id) REFERENCES User;
19
20  CREATE LOCAL INDEX PhotosByTime
21    ON Photo(user_id, time);
22
23  CREATE GLOBAL INDEX PhotosByTag
24    ON Photo(tag) STORING (thumbnail_url);
```

Sample schema for photo sharing service (Source: [Megastore] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services." Proceedings of the Conference on Innovative Data system Research (CIDR) (2011), pp. 223-234)

There are two distinct kinds of tables in Megastore entity groups: root tables and child tables. Each child table should indeed specify a single distinct foreign key referring to a root table, as shown by the `ENTITY GROUP KEY` annotation at **line 18** in the schema above. Every child entity refers to a specific entity in the root table, which is known as the **root entity**. A root entity and all entities in child tables

referencing to root entity make up an entity group. As per the Megastore paper, a megastore instance can have many root tables, leading to different classes of entity groups.

In our example, each user's photo collection is a distinct entity group in the aforementioned schema. The `User` in **line 4** is the root entity, while the `Photo` in **line 10** are child entities. By repeating the `Photo.tag` field, we can attach multiple tags to a single photo without creating a new table.

Pre-joining with keys

In traditional relational database design, it is suggested that primary keys have surrogate values to identify each row in a table. However, in Megastore, the entities that are likely to be read together are grouped and assigned a key. This allows for more efficient data retrieval and access. Each entity is assigned to its own Bigtable row. The Bigtable row key is created by combining the primary key values. All properties other than the primary key have their own column in Bigtable.

Indexes

We can define secondary indexes on the list of entity property and fields in protocol buffers. We differentiate two types of high-level indexes:

- **Local index:** Each entity group's local index is used as a distinct index. Data inside an entity group can be located using local indexes. They are organized and maintained in entity groups such that the updates are made to the entire group consistently and atomically. In the schema above, `PhotosByTime` in **line 20** is a local index.
- **Global index:** A global index can encompass many entity groups. It is used to locate entities without knowing which entity groups include them in advance. In the above schema, `PhotosByTag` in **line 23** is a global index.

The additional features in Megastore indexing are as follows:

Storing clause

We can save more properties from the primary table for quicker retrieval during read operations by adding the storing clause to an index. For example, the index of `PhotosByTag` saves the thumbnail URL of the photo for quicker access.

Repeated index

It allows for indexing the repeated properties, e.g., `PhotosByTag` is a repeated index.

Inline indexes

Inline indexes are helpful in extracting information from child entities and storing it in the parent for quick access. They are also useful for implementing many-to-many relationships more effectively.

Mapping to Bigtable

The table and property name of Megastore is appended to make the Bigtable column name. This avoids collisions when mapping entities from distinct Megastore tables into the same Bigtable row. The table below depicts how the data of the photo-sharing service above will be arranged in the Bigtable. (Source)

Megastore Data in Bigtable

Row Key	User Name	Photo Time	Photo Tag	Photo URL
101	John			
101,500		10:30:05	Dinner, Paris	...
101,502		11:30:06	Betty, Paris	...
102	Mary			

- The row of Bigtable for a root entity stores the metadata of transaction and replication for the entity group. It also stores the transaction log.
- Metadata is in the same row, which allows updating atomically through a single Bigtable transaction.
- A single Bigtable row represents each index item. The cell's row key is created by concatenating the indexed property values with the indexed entity's primary key.

Transactions and concurrency control

An entity group within a Megastore works as a small database with serializable ACID (atomicity, consistency, isolation, durability) properties. In a transaction, changes or mutations are recorded in the write-ahead log of the entity group. The changes are then applied to the data.

Bigtable gives the capability to store several values with various timestamps in the same row/column duo. Megastore uses this feature to implement multi-version concurrency control (MVCC). When mutations are executed within a transaction, the values are recorded at the transaction's timestamp. To prevent seeing incomplete changes, readers utilize the timestamp of the most recently fully applied transaction. Readers and writers do not interfere with one another, and reads are separated from writes during a transaction.

Three levels of reads consistency

Megastore provides the following levels of read consistency:

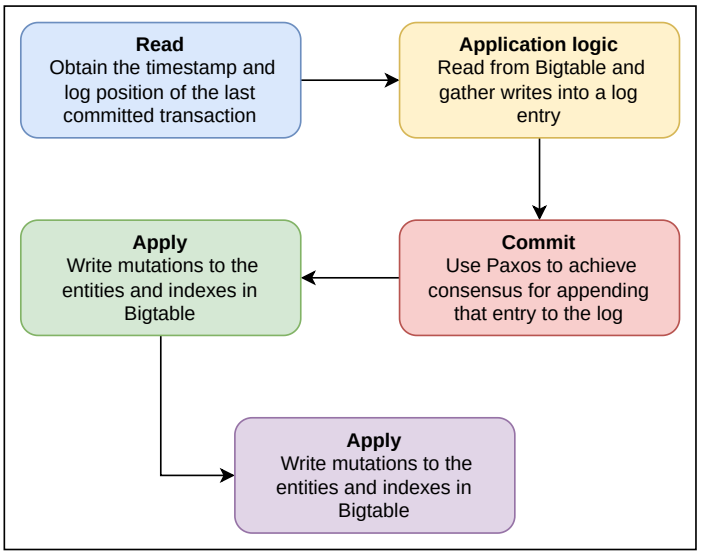
- **Current:** This makes sure that all previously committed writes have been executed. After that, the application reads from the most recent committed transaction's timestamp.
- **Snapshot:** This reads the latest committed write operation.

- **Inconsistent:** This reads the most recent value without considering the log's state.

Writes

To find the next available position of the log, a current read is executed before every write transaction. The commit action compiles all the changes into a single log entry, allocates a timestamp that is greater than the preceding timestamp, and then concatenates it to the log via Paxos. The complete write lifecycle is as follows:

> The write cycle below is derived from the Megastore research paper.

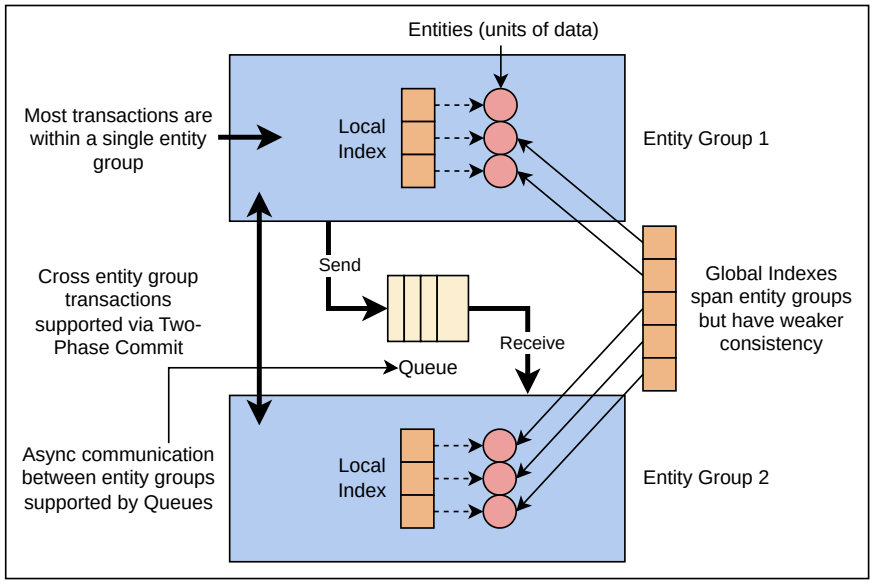


Writes lifecycle

Although the write operation makes a great effort to wait for the closest replica to apply, it can revert to the client at any time after the commit.

Queues

The transactional messaging among groups of entities are handled through queues. They can be used for inter-group operations, batching many changes into a single transaction, or deferring work. A transaction on an entity group may atomically receive or send more than one message in addition to modifying its entities. Every message has a single entity group for both sending and receiving. If they vary, delivery is asynchronous, as shown in the illustration below:



Operations between entity groups

Queues provide a method for performing actions that affect several entity groups.

Two-phase commit

For atomic changes across entity groups, Megastore enables a two-phase commit. They can aid in the simplification of the application code for unique secondary key enforcement. However, queues are preferred over the two-phase commit, since these transactions have substantially higher latency and raise the danger of conflict.

Other features

Following are some other features that Megastore offers:

- **Integrated backup system:** It is used to restore a group of entities to a state that lies at any point in past time.
- **Data encryption:** Applications can encrypt data at rest, along with transaction logs, if desired. Encryption employs a unique key for each entity group.

Let’s dig deep into the replication scheme of Megastore in the next lesson.

← Back

Next →