Development Report

Cassandra Database

Player sessions service supposed to perform two types of queries:

a) Fetch “start” sessions for the last X hours for each country;
b) Fetch the last 20 “end” sessions for a given player_id.

We are going to store all the data associated with service in *player_events* keyspace.

```
CREATE KEYSPACE IF NOT EXISTS player_events
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'}
    AND durable_writes = true;
```

Designing structure of the tables we should take a query first approach. A *start_sessions_events* table serves for type (a) queries and an *end_sessions_events* table serves for type (b) queries.

To query *start_sessions_events* table efficiently we need to partition the table by *country* and order data in each partition by *ts* in descending order. To ensure that primary key is unique, we add *session_id* in primary key.

```
CREATE TABLE IF NOT EXISTS start_sessions_events (
    country text,
    ts timestamp,
    session_id text,
    event text,
    player_id text,
    PRIMARY KEY (country, ts, session_id)
) WITH CLUSTERING ORDER BY (ts DESC, session_id ASC);
```

Assuming that *relevance_datetime = now() – X hours* we can fetch information from the table this way:

```
SELECT JSON *
FROM start_sessions_events
WHERE ts >= relevance_datetime
ALLOW FILTERING;
```

Despite we use ALLOW FILTERING which is not recommended, this approach helps us to fetch recent sessions grouped by *country* in the most efficient way. Cassandra will explore partitions (countries) one by one and filter records by sorted key *ts* within each partition. That means Cassandra should read only necessary records without mixing countries in output.

To query *end_sessions_events* table efficiently we need to partition the table by *player_id* and order data in each partition by *ts* in descending order. To ensure that primary key is unique, we add *session_id* in primary key.

```
CREATE TABLE IF NOT EXISTS end_sessions_events (
    player_id text,
    ts timestamp,
    session_id text,
    event text,
    PRIMARY KEY (player_id, ts, session_id))
    WITH CLUSTERING ORDER BY (ts DESC, session_id ASC);
```

With this table structure it’s quite simple to fetch data:

```
SELECT JSON *
FROM end_sessions_events
WHERE player_id = 'cd32jdv4y5je4djnjs4alv'
LIMIT 20;
```

All records associated with a given *player_id* are stored in one partition and already sorted by *ts* desc. The query returns top 20 records from this partition.

We decided not to use batch insert, because records in a batch usually belong to different partitions. This may decrease performance of a POST method.

Rest API Server

Server side of the application connects to the database and initializes keyspace and tables if they have not been initialized yet. For each get and post method there is a separate class.

```
class PutSessions(Resource)
class EndEventsByPlayer(Resource)
class RecentStartEvents(Resource)

api.add_resource(PutSessions, '/put_events')
api.add_resource(EndEventsByPlayer, '/end_players_events/<string:player_id>')
api.add_resource(RecentStartEvents, '/start_players_events/<string:hrs>')
```

It makes application easy to extend. Developer should just add new database objects and new classes to create rest api methods.

Body of POST request is passed as a json-object, which is easier to validate than arbitrary text.

We use flask to create prototype of the service, because it's easy to use, extensible and supported by large community of developers. For production application we should consider more secure and scalable module (e.g. Django).

Future Improvements

- Set proper replication factor and cluster topology while creating keyspace on production Cassandra cluster
- Authentication and TLS
- Schema validation. Store invalid events in special table and notify client about failure
- Asynchronous request processing
- Paging in case of large output
- Multiple instances of the service and load balancing in case of high RPS (and to avoid single point of failure)
- Optionally, we may restrict RPS from each client