Self-evaluation

- HTML: 4
- Javascript: 3
- CSS: 3
- Java: 4.5
- Python: 4.5
- PostgreSQL: 3
- Couchbase: 1
- ElasticSearch: 3.5 (although I'm pretty good with Lucene internals, I'd give myself a 4.5 there)
- SQL: 3.5
- Nginx: 2
- Node.js: 2
- Scala: 2
- Docker: 2.5
- Kubernetes: 2
- AWS: 4
- Chef: 1
- Ansible: 1

Top Email Domains

(this is a copy of the readme file of the project - please check out the source code as well)

Requirements

To run this program, you need: - Java version 1.8 (or higher)

Compiling and running

This project uses `gradle` and you can run tasks with the included wrapper.

Generate runnable artifacts and run the application:

```
$ ./gradlew installDist
$ cd build/install/kahoot-top-email/bin
$ ./kahoot-top-email
```

By default, the application will read from standard in until it encounters the string `end`. You can read from

a file by supplying the following arguments:

```
$ ./kahoot-top-email --source=file --file=/path/to/file/name
```

You can also read from a jar resource by supplying the following arguments:

```
$ ./kahoot-top-email --source=resource --resource=emails.txt
```

Running tests (from the root project folder):

```
$ ./gradlew test
```

Kahoot Robo Parts

Summary

REST interface implemented with the sparkjava mini framework.

Database has a simple structure with 2 tables:

- `parts`
  - `id` : int
  - `name` : string
  - (I didn't implement the other fields, because the logic is not much different from the name)
- `compatibilities`
  - `part_id_1` : int (foreign key to `parts.id`)
  - `part_id_2` : int (foreign key to `parts.id`)

The `compatibilities` table is a many to many association that associates `part` entities with each other. Through this table we can resolve compatibility related queries.

The API contains 2 implementations for the database-related logic: - Memory based: useful for testing - all objects are kept in memory - Hibernate based: using `hibernate` ORM to manage the DB

The client is implemented with react and redux (see `./client` folder)

Documentation is written in swagger (see `./docs` folder).

How to run

Run backend:

```
$ gradle run
```

Run client:

```
$ cd client/react_client
$ npm run start
```

API documentation:

```
$ cd docs/html
$ open index.html
```

Scalability and high-availability issues

Database

This API is a simple access layer on top of a database, so under heavy load the bottleneck will be the database.

The database can be scaled vertically (bigger server) or horizontally (master-slave replication). Under a master-slave replication, application servers can be redirected to read from different slaves, thus spreading the read load. Writes will have go through the master, but this probably less of a problem since writes will be less frequent than reads.

Alternatively, if there is a high write throughput, we could use a more high-performance database, such as Apache Cassandra.

Application + Deployment

The application should run in a container (through Docker) and then deployed with a container management service (such as Kubernetes or Amazon ECS). This service can easily take care of replication and scalability.

If deployed in AWS, we would probably use ECS, The application servers should run under a load balancer.

Security

In this implementation, security has not been addressed, but in a real life scenario we would use one of the following strategies (or probably both):

- Secure access at an application level through something like OAuth
- Secure access at an infrastructure / network level (for example, have the API run in a private network, or in a closed security group) and allow only pre-determined clients to access it

Click-through rate

According to similarweb rankings, kahhoot.com receives around 7 million visits per month. That is about 230000 per day, which amounts to 9600 visits per day, and is roughly equal to 3 visits per second.

My assumption is that this number will grow by at least a factor of 1 (70 million visits per month) or, preferably, a factor of 2 (700 million visits per month = 300 visits per second). We'll keep this number in mind: **300 visits / second**. We'll use it as a performance lower bound for the rest of the analysis. Whatever system we build, it has to perform at least this good and it should be easy to scale up with predictable effort, preferably just by adding extra hardware and not changing anything fundamental about the infrastructure.

What to store

In order to compute the click-through rate (CTR), we should store every action done by the users on the website and then compute the business metric by running aggregation queries on the event store.

Each action has a particular type. For example:

- `VISIT_HOME_PAGE` - triggered when a user first lands on the home page
- `SEARCH_FOR_ITEM` - triggered when a user searches for an item
- `CLICK_ON_SEARCH_RESULT` - triggered when a user clicks on a search result
- etc...

All actions have some common attributes, such as:

- `SESSION_ID` - the unique identifier of the user's session
- `TIMESTAMP` - timestamp representing the date and time when the event took place
- `ORDER` - the order of the event within the user session
- `USER_ID` - the unique identifier of the user
- etc...

Depending on the type of the action, we will store additional information. For example, for the `SEARCH_FOR_ITEM` action, we will store the exact search query and the result list delivered by the system. For the `CLICK_ON_SEACH_RESULT` action, we will store the `id` of the item clicked on, the position of the item in the result list, etc...

Once we have the proper event store system in place, computing the click-through rate is a simple query. For example, given a time interval `T = (START_TIMESTAMP, END_TIMESTAMP)`, and a kahoot item `KAHOOT_ID`, we can compute the `CTR` using the following pseudo-SQL queries:

```
select clicks.count / impressions.count
from
(
  SELECT COUNT(*) as count
  FROM events
  WHERE events.type = SEARCH_FOR_ITEM
  AND events.result_list INCLUDE (KAHOOT_ID)
  AND events.timestamp > START_TIMESTAMP
  AND events.timestamp < END_TIMESTAMP
) impressions
join
(
  SELECT COUNT(*) as count
  FROM events
  WHERE events.type = CLICK_ON_SEARCH_RESULT
  AND events.item_id = KAHOOT_ID
  AND events.timestamp > START_TIMESTAMP
  AND events.timestamp < END_TIMESTAMP)
) clicks
```

We can further segment this query by adding additional search parameters. For example, we might want to segment on user age, gender, etc.

Where to store

We need a solution that supports the heavy write-load throughput as well as enables writing analytics queries on a huge quantity of data.

To solve the write load, we can use the following 2 open source systems:

- Apache Cassandra
- Apache Kafka

Cassandra is a no-sql distributed database. Kafka is an distributed log / event store.

By design, both systems perform excellently under heavy write load and scale linearly with the number of nodes in the cluster. The difference between them is that Cassandra is a database (i.e. - it allows update and delete operation on its data), whereas Kafka is an append-only datastore (it doesn't allow updates or deletes, only inserts).

A Cassandra based solution would allow for writting analytics queries directly on the Cassandra data,

without further transformations, although we do have to make sure that the data in stored in a queryable way. Cassandra is bad with secondary indices, so when we insert the data, we need to do so in a way in which is easy to query.

A Kafka based solution would need an extra system that reads the data from Kafka and stores it in a queryable format. A popular solution is to dump the Kafka data into an S3 bucket (or a HDFS cluster), and then use Apache Spark batch jobs to derive the required metrics.

The final metric (CTR) should be stored in a traditional relational database engine (mySql or postgre, preferably also behind an in-memory cache) in a format which is easy to query by the search index.

Data availability

The click-through rating seems to be a relatively stable metric, so it would probably be OK to use data which is a bit old (say 1 day old).

If we use Cassandra, then, the delays should be small, since we can directly query Cassandra for the data we need, and, in theory, the data is available shortly after it is written.

If we use Kafka, then the delays are bit larger, because we would have to first store the data in a file system and then process it through a batch job.

However, it is possible to achieve minimal delays in a Kafka based system. We could do so by programming a Kafka listener that updates a cache as as it sees a new event arriving on the stream. This cache is queryable by the search index.

Closing summary

My choice for this system would be:

- Store all the actions that user do on the website or other clients (mobile apps, etc.) as events
- Route all the events through Kafka
- Have Kafka dump the events on S3
- Run Apache Spark jobs that read the S3 dumps and compute fine-grained metrics (use AWS Elastic-Map-Reduce clusters for this)
- Store these metrics in an easy-to-query databases (Redis, MySQL, Postgres) that make them avaiable to the rest of the organization
- If we need minimal delays, write special components that plug into the Kafka stream and compute the required data

Additional samples of work

Joni Dep - Simple Java Dependency Injection

I've written a very simple dependency injection framework for Java, called Joni Dep, available on Github.

Technologies you should look into

- Apache Kakfa - distributed event store
- Amazon EMR (elastic map reduce) - cheap and easy way to start computing clusters for batch jobs
- Kotlin - a Java derived language with much needed syntactic niceties
- Apache Spark - great map reduce tool - awesome for big data batch job computations