### Introduction

Thanks to our early engineering team, which cut their teeth building large-scale systems at Uber, Bigeye’s infrastructure was thoughtfully engineered to handle scale from the get-go.

Recently, this infrastructure was put to the test when a partner connected a data warehouse with thousands of schemas and 50,000 tables to Bigeye – a number that is more than 10 times what we usually expect. To our relief and pride, Bigeye handled all those tables like a champ, apparently, the only product the partner tried out that didn’t fall over under the load.

In this blog post, we will give a bit more color on Bigeye’s infrastructure, and the engineering best practices that have been implemented to make it a robust (and cost-effective, for customers) product.

### Bigeye architecture

<architecture image>

As shown in the architecture diagram above, Bigeye consists of a few services:

* **Datawatch** - core monolithic app that goes to connect to the customer warehouses and actually does the work
* **ML platform** - comprises a training layer and a serving layer that performs the autometric and autothreshold computation
* **Scheduler -** that goes in anything that is supposed to run at a particular time
* **AWS RDS (MySQL)** database with a **Redis** caching layer
* **RabbitMQ** - queueing tasks

### How Bigeye works

When you first connect a new data source, Bigeye sucks in all the schemas and tables in the data source. The profiler, which is a component of Datawatch, then goes through each of the columns in each of the tables. If you’ve set a row\_creation time, it queries two days of history with a limit of 10,000 rows. If you haven’t set up a row\_creation time, it’s just the select \* limit. The profile then applies a set of heuristics to the data to determine which autometrics to suggest for the column.

Some examples of the heuristics:

* + If you’ve got three values, it’s probably an enum
  + If you have no duplicate values, maybe you never want them to be duplicates
  + Maybe it looks like an ID column, which means you’ll want to check for duplicates
  + If the column is full of strings, there are a bunch of tests that are run, to detect maybe dates, maybe ID

You can select only a subset of schemas to profile, but regardless, this process is generally fairly expensive. Luckily, you really only have to profile once up-front for every new table (columns that change are also re-profiled).

### The usual chokepoints

In the profiling flow above, there are a couple of weak points that, if improperly designed, can result in the service having problems.

The first is the customer’s data warehouse, and the risk that we issue so many queries that we overwhelm them. The second is Bigeye’s own service database and local store. There are, of course, also theoretically restrictions on the number of API calls and things like that, but this will largely be a function of the limits on the databases.

### Things Bigeye does that are best practice

Bigeye has been deliberately designed to minimize pressure on these chokepoints. In particular, the initial profiling of tables is not done all at once synchronously. Rather, it’s broken into chunks and done asynchronously. This also results in a better user experience – by making these heavy requests asynchronous rather than synchronous, the user doesn’t have to sit there and wait for Bigeye to index and profile every single table before they see some interaction in the UI.

What does this mean in more detail?

It means that the tables aren’t just all placed in the queue at once. Rather, each table (or set of columns) is scheduled for an arbitrary time throughout the day. This ensures that the Bigeye doesn’t overwhelm the customer’s data warehouse or take up too much available computing power.

To process the tables asynchronously, Bigeye uses a **message queue**, specifically RabbitMQ. RabbitMQ provides an efficient way to store pending transactions (in this case, requests for tables to be processed) and forward them on to somewhere else that handles them. (ELI5 tip: you use RabbitMQ whenever your service is doing something that may a significant amount of time, like uploading large files, sending large emails, video encoding, etc).

Another best practice that Bigeye implements is connection pooling. Ordinarily, every time you connect to a database, you open and close a connection and a network socket. This opening and closing takes compute power. Most of the time, for simple operations, this is not very expensive, but as things scale up, it can impact performance.

Often, it instead makes sense to “pool” the connections, keeping them open and passing them from operation to operation. Bigeye does this when connecting to the data warehouses with JDBC APIs. (Pooling is provided as a [standard feature](https://www.baeldung.com/java-connection-pooling.) from Java libraries). We also configure the JDBC connection pool to accept only a certain maximum number of connections – this causes the app to block a bit waiting for a new connection, which we use to provide backpressure to prevent overloading of the customer warehouse.

### How Bigeye manages warehouse costs for clients

There’s another component to all of this that Bigeye is very conscious of. Bigeye’s product runs on top of data warehouses (querying data warehouses), and most data warehouses are charging by usage now. To prevent a situation where a customer is stuck with a huge cloud warehouse bill, Bigeye tries to be very judicious about its querying, in particular:

* Storing the data as it comes back in its own database – unless you’re going to the preview page, Bigeye is not live querying data.
* Batching metric queries together; for instance, rather than running three separate queries for a max, a min, and an average, running one query for all three keeps the cost down.
* Being aware of warehouse particularities and working around them. For example, since Snowflake charges customers based on how long the warehouse is running, Bigeye tries to run all the Snowflake queries at the same time, rather than say, every ten minutes, which would keep the customer’s instance up all night, even if it’s technically “spread out”.