In order to provide this data to millions of concurrent users, we would want to be able to automatically scale out our web application across many nodes dynamically using an orchestrator like Kubernetes. By automating the deployment and scaling of the application, we would be able to ensure that we are keeping up with request demand while minimizing our own resource consumption costs.

While the Socrata API doesn’t currently impose throttling limits on requests that use an application token, they ask consumers to be mindful by keeping usage as efficient as possible. They reserve the right to change this policy in the future. So any design should keep calls to this API as minimal as possible.

In order to keep Socrata API calls to a minimum, I would keep a copy of the API data in our own data store that we would refresh. We would need to be able to replicate changes to the data store across many other nodes in our cluster. We know that none of the users of our system will be modifying data. So we just need to ensure that our design can scale with a lot of reads.

To keep our copy up to date, one idea would be to have a dedicated service(s) that would periodically check for changes in the source data, and then write these changes to our own copy. Any new updates could also be distributed across all of the other nodes by using an event stream like Apache Kafka. All other nodes would subscribe to this stream and update their own copies as new events are published by the writer. We would also build in the ability for a subscriber to replay events it might have missed because of error conditions by sending a request to the publishing service upon recovery.

We would still need a way to constrain the writer service to only making calls to the Socrata API as needed. My first thought would be to drive the frequency of writer-to-Socrata calls based on read request frequency. Since Kafka was built to handle high volume web activity, all of our reader nodes could publish events to a topic stream that our writer service would consume. As long as the writer continued to get regular events, it would continue to refresh. If there were periods of inactivity, the writer wouldn’t make any unnecessary API calls. We would put a cap on the refresh frequency (i.e. once per minute) initially, and we could further fine-tune this frequency based on observations of how often the API data changes since that is an unknown.

By keeping our own copy of the data that remains fresh, we also gain the ability to employ many other read-specific optimization patterns such as materialized views to ensure that our interfaces are meaningful and responsive.