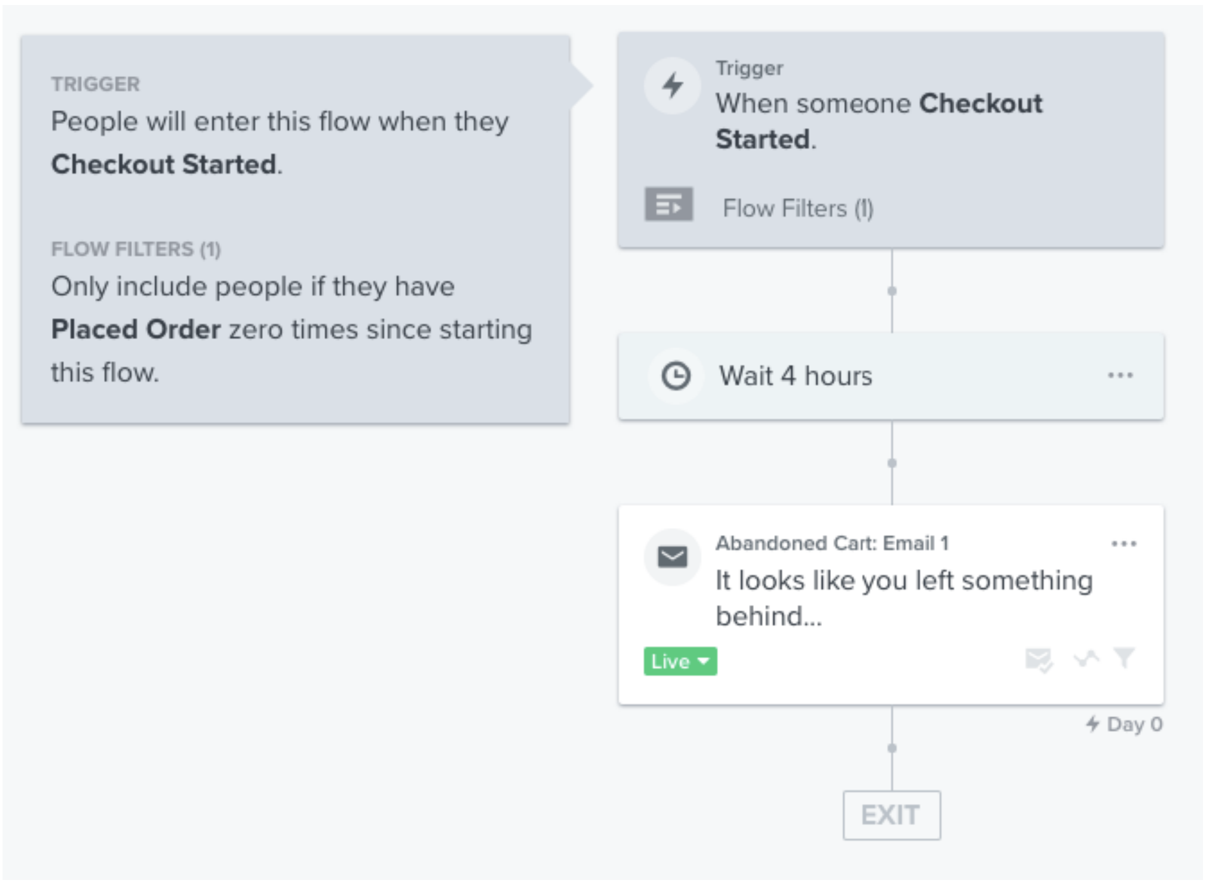
Moving to Batched Workloads

Author: Josh Panka

Claps: 303

Date: Jan 29, 2019

Before getting started on the technical benefits of batch processing, it is helpful to understand what the event automation pipeline is within Klaviyo. At Klaviyo the event automation pipeline is referred to as flows, and the basic idea of a flow is to automate emails or other actions based on any event received from our users. Below is an example of a basic flow a user can build within Klaviyo. The given flow sends an email to a given companyâ€™s customer if they have started a checkout and have not placed an order within a four hour window.



TRIGGER
People will enter this flow when they **Checkout Started**.

FLOW FILTERS (1)
Only include people if they have **Placed Order** zero times since starting this flow.

Trigger
When someone **Checkout Started**.
Flow Filters (1)

Wait 4 hours

Abandoned Cart: Email 1
It looks like you left something behind...

Flows, like the one pictured, create evaluations to perform an action at a future point in time. In the case of the this flow, after someone starts a checkout an evaluation is created to send an email after four hours. After four hours the execution pipeline processes the evaluation, and if the individual has not placed an order, he or she receives the abandoned cart email.

Because Klaviyo processes upwards of a billion ecommerce events a day and our customers have created tens of thousands of flows we are very invested in automating the efficiency of flows to improve both the user experience as well as our system performance. A lot of the recent improvements to our system performance have come from the addition of batch processing.

Benefits of Batch Processing

Immediate Improvements

When items within a workload are similar, and not latency-sensitive, having the ability to buffer and aggregate the items in order to perform fewer queries leads to better performance. Flows are a perfect use case for batching because, per the above example, itâ€™s easy to see that many evaluations get created to send a common email. The immediate improvement obtained from batch processing is the ability to cluster queries based on the specific email or action. For example, a group of evaluations for the aforementioned email action are set to go out at noon. If we processed each evaluation individually every send would result in:

1. Checking the actionâ€™s status.
2. Querying for the associated email
3. Rendering the email.

These donâ€™t sound like expensive queries or actions, but at scale even delays of 1â€“2ms can turn into additional seconds spent processing. A lot of the performance gains batching makes available are from reducing the amount of little things that are done repeatedly. For instance, limiting the number of network round trips or limiting the number of generated query plans are little things that add up and create delays. Assuming that the execution pipeline has a batch size of 500 evaluations and 1 million evaluations are set to be executed for a given email, then instead of issuing 1 million queries to check an actionâ€™s status we only need 2,000 queries to check a given actionâ€™s status.

Implementation Improvements

Outside of just getting the immediate improvements from batching on a common action it is easy to miss that one might be querying for specialized information on a per evaluation basis. The querying for this specialized evaluation information individually is referred to as the N +1 query problem.

An example of the N + 1 query problem is that if we have a batch of evaluations all with associated customers we could perform that work in a way where we do a bulk query for the evaluations and then individually query for all of the customers associated with each evaluation. Performing a bulk query and then individual queries for customer information causes 1 query to occur for the evaluations and then N additional queries for the individual customers. Below we can see a contrived example of the N + 1 problem.

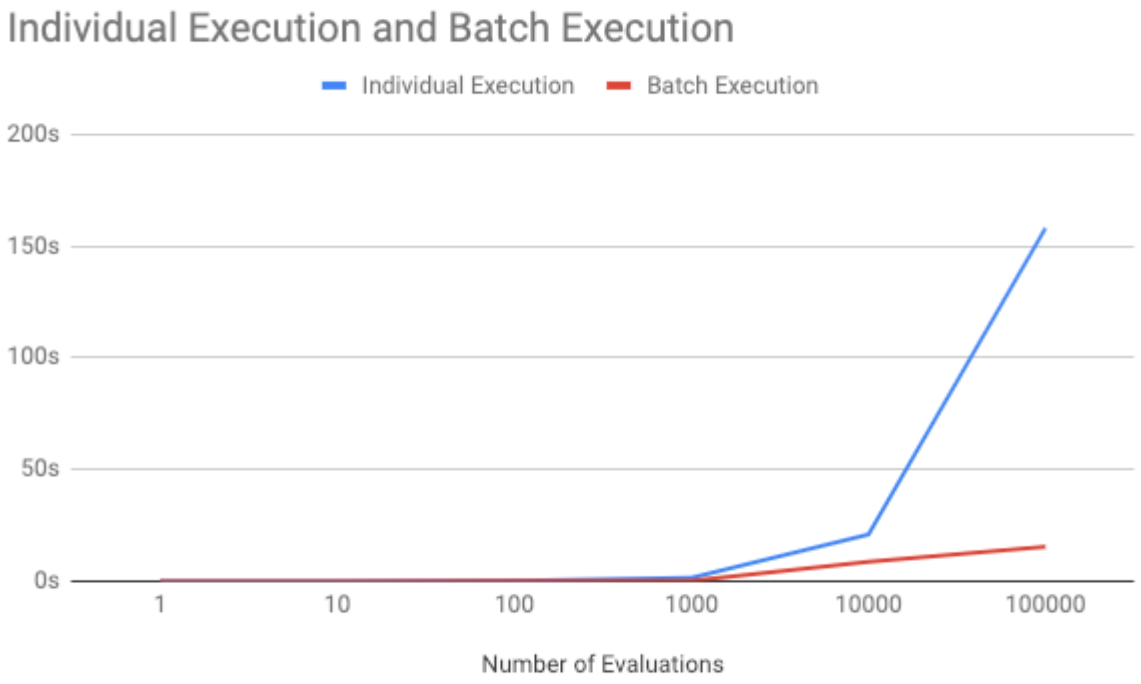
```
evaluations = SELECT * FROM evaluation WHERE id IN evaluation_ids
for evaluation in evaluations:
 customer = SELECT * FROM customer WHERE id = evaluation.customer_id
```

The above example works but leaves a lot on the table in terms of performance gains. Instead, it would be better to do a bulk query for all of the customers associated with each evaluation right after we perform the bulk query for the evaluations. An example of this can be seen below.

```
evaluations = SELECT * FROM evaluation WHERE id IN evaluation_ids
customer_ids = [e.customer_id for e in evaluations]
customers = SELECT * FROM customer WHERE id IN customer_ids
customers_by_id = {c.id: c for c in customers}
```

```
for evaluation in evaluations:
 customer = customers_by_id[evaluation.customer_id]
```

Now we are performing 2 queries instead of flooding our database with N + 1 queries. Assuming a batch size of 500, the performance improvement between the individual and batch execution is roughly 10x when processing over 1000 evaluations. The results can be seen in the graph below:



Things to Keep In Mind

Unsurprisingly, reducing the number of queries results in a lot of speed improvements. However, as code was refactored to accommodate for batch sending the general complexity of our application increased. For us, the benefits greatly outweighed the added complexity of moving to a batched flow execution, but with the added complexity came a few extra things we needed to keep in mind. The main added complexity can be attributed to the following:

- Retryable errors need to be handled in a more granular way.
- More managed assets means more potential failure modes.
- Adding intermediate processing states changes how failures are handled.

When working with errors in a pipeline that focused on individual evaluations, if a retryable error occurred we could immediately retry the error. In the world of batch execution, more care needs to be taken into account to make sure we are only retrying the failed evaluations and not the entire batch. Handling the errors in an individualized way is not a huge deal but is necessary to ensure that the only work that is retried is the failed work.

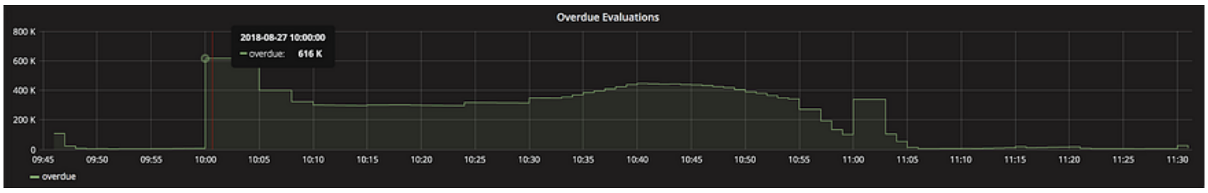
In order to batch flows we added a Redis cluster that buffers work at a one minute interval. For instance, if we wanted to batch based on an action_id, the bucket key would become action_id and the values would be the individual evaluation ids. Adding Redis into the mix adds an extra step to the flow execution pipeline. Adding an extra step means additional understanding of how to manage the failure modes and how to recover gracefully. Once again this is not a huge deal, but it will increase future ramp time for new engineers and increases the number of failure modes that flows can experience.

Finally, intermediate states were added to the flow execution pipeline. Adding intermediate states means that evaluations are no longer just one of two states: waiting or completed.

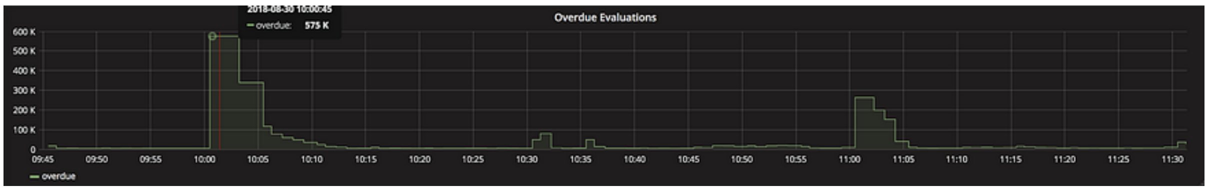
By adding intermediate states to a system we greatly reduced the amount of duplicate work in the pipeline; however, doing this introduces ways in which an evaluation could exit flow execution in an unhappy way (usually through runtime errors). Adding intermediate states to any system requires much more thought in order to keep the general flow execution state machine in working order.

Conclusion

In the end, adding batch execution to flows greatly increases the performance and sets up our system for continued success. Graph 1 and 2 show that adding batch processing to flows reduces the time to process large sends by an order of magnitude.



Graph 1 â€” Top of the hour send prior to batching.



Graph 2 â€” Top of the hour send while batching.

The improved performance means happier customers and a >50% reduction in the number of EC2 instances. This improved performance does not come without added technical concerns around failure modes. Overall, flows can now process work much more efficiently and effectively, but like any additive engineering change it comes with added complexity.