Magic: Constant-time tricks that Klaviyo uses to operate at huge scale

Author: Peter Gaivoronski

Claps: 27

Date: Feb 13, 2020

Klaviyoâ€™s data storage and processing needs are vast and have rapidly increased over time. Our external end-users and internal service consumers do not care about our data size, however, and still need rapid responses to their queries. To keep our systems running smoothly, we occasionally have to employ various unconventional optimization techniques. The specific techniques mentioned in this article are ways of taking what are normally linear or even more complex operations and making them constant time.

To begin, what do we mean by constant time? The Wikipedia definition of constant time complexity is

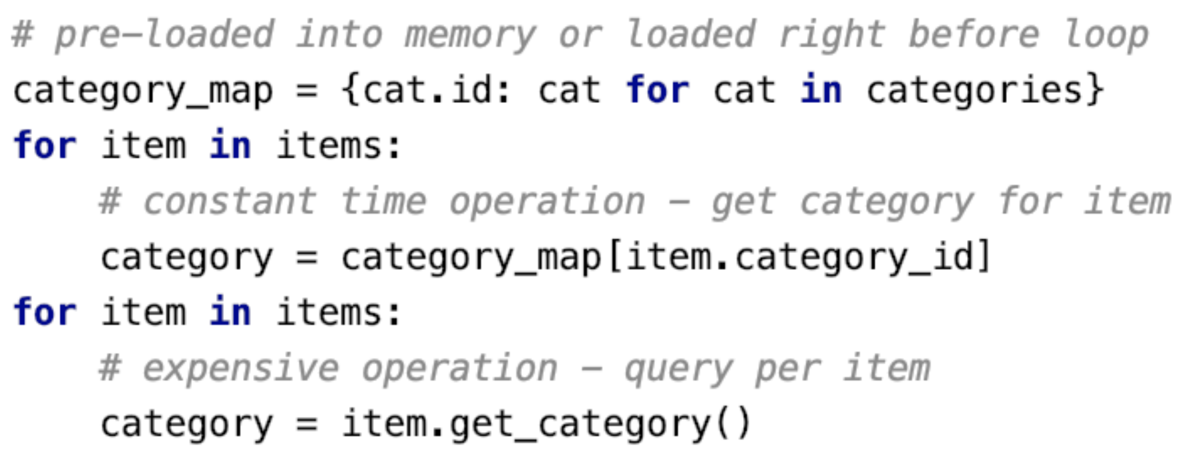
> If the value of $T(n)$ is bounded by a value that does not depend on the size of the input. For example, accessing any single element in an array takes constant time as only one operation has to be performed to locate it. In a similar manner, finding the minimal value in an array sorted in ascending order; it is the first element. â€” Wikipedia

This is a great general definition, but I would like to propose a more specific one when talking about the kinds of constant time operations that Klaviyo is interested in. A constant time operation, then, is an operation which at the point in time that we want to know some value, takes a short and roughly similar amount of time to find out the answer, no matter how large the relevant dataset is.

Constant time operations are not guaranteed to be fast according to the general definition, just constant. It is possible to have an operation that takes the same amount of time for any dataset but takes a very long time to perform overall. Such an operation can be considered constant time, but it is not the kind of operation that we are interested in. Since we are trying to optimize our code and return values quickly, we are interested in constant time operations that complete very quickly, almost instantly as far as the end-user is concerned. In many cases, this requires an asynchronous non-constant time component running in the background that makes the magic possible. Like any magic trick, there is something going on behind the scenes that the audience is not completely aware of.

In-Memory Mapping

One very simple optimization that is familiar to many software engineers is the â€œHuge In-Memory Mapâ€�. This is simply a large collection of keys that can be used to quickly retrieve values. Its magic consists of allowing quick extraction of data and, in the case where we are doing data processing, only querying as much data as is needed for the task. The general use cases of in-memory maps include loading auxiliary data before a loop or holding global values in an easy to access place. For example, if we want to process a set of items and each item has a category, fetching the category for every item may be expensive, and would certainly not be constant time with regard to the number of items. It would be linear at query time, because the query time would at least indirectly depend on the number of categories in the database through table index size, for example. We can make this constant time by loading all the categories in memory before processing the items, and then referencing each category directly from each item. The in-memory map pattern, illustrated in figure 1, allows us to get the category in constant time for each item.



```python
# pre-loaded into memory or loaded right before loop
category_map = {cat.id: cat for cat in categories}
for item in items:
    # constant time operation - get category for item
    category = category_map[item.category_id]
for item in items:
    # expensive operation - query per item
    category = item.get_category()
```

Figure 1

When is a good time to use the huge in-memory map? Any time we want to â€œlook upâ€� a value by some key during processing, especially if the auxiliary data is easy to map to our processing dataset. While simple, this pattern can also be dangerous because the number of things we can load into memory is highly dependent on the free memory we have at any given time. If we load too much data into memory and donâ€™t clear any of it, we risk an Out Of Memory condition where the process can get killed, or at least a Paging condition where the disk will be heavily loaded as our program tries to get relevant data off the disk and into memory whenever it runs, slowing everything down considerably.

In many of these techniques we will see that our applicationâ€™s ability to deal with stale data is extremely important. When designing systems for performance, it is vital to keep track of what can be stale where, because many times we can optimize things merely because they do not need to have up to date data at every invocation. For example, for the in-memory map we have to make sure that our map is up to date, or that our use case can tolerate the map being stale. If the map does not have the value, we will not be able to find it during processing. However, that may not be important because we can fetch the item if it is missing. The value may also be in the map but outdated. In this case, it would depend on whether our application can simply use the outdated value without a problem, or whether we want to keep track of when each item was added to the map and ignore or refetch values that are older than some certain value.

Another interesting constant time operation is looking up values using a â€œBloom Filterâ€�. A bloom filter is a probabilistic data store that is highly space efficient. We can load a vast amount of information into a small binary blob and then filter against it. The primary use case is to reduce the search space for loops, because the bloom filter can tell us with full accuracy whether an element is not in a set, or with configurable accuracy whether an element might be in a set. The accuracy of a bloom filter depends on its size, which is determined ahead of time using a formula that takes the max number of elements the filter allows and the target accuracy of that filter. Theoretically we can put more than the max number of elements into a given bloom filter, but the accuracy will decrease as a result.

The best way to use a bloom filter is to load a gigantic amount of information into it, an entire set for example, in an asynchronous linear time process. We can keep refreshing the filter on a certain time interval, ensuring that our data is not very stale. When matching a different set against this filter, we can get a constant time decision on whether the element is in not in the target set or might be in the set. If it is possibly in the set, then we can perform our expensive operation to verify whether it is in the set or not. So when we are looping through our set we have an almost constant time performance instead of a linear time performance that depends on the size of the filtering set.

At Klaviyo, we often use bloom filters when we compare email lists against each other. It is fundamentally a linear time operation to say whether a specific email in list A is also in list B, which depends on the size of list B. However, if list B can be represented as a bloom filter, then we can be sure whether that email is not in list B or might be in list B in constant time. If the email might be in list B, we are over 99% sure that it is, but just in case we do a linear check. In the worst case this is still linear time, but most of the time it ends up being constant time due to low overlaps.

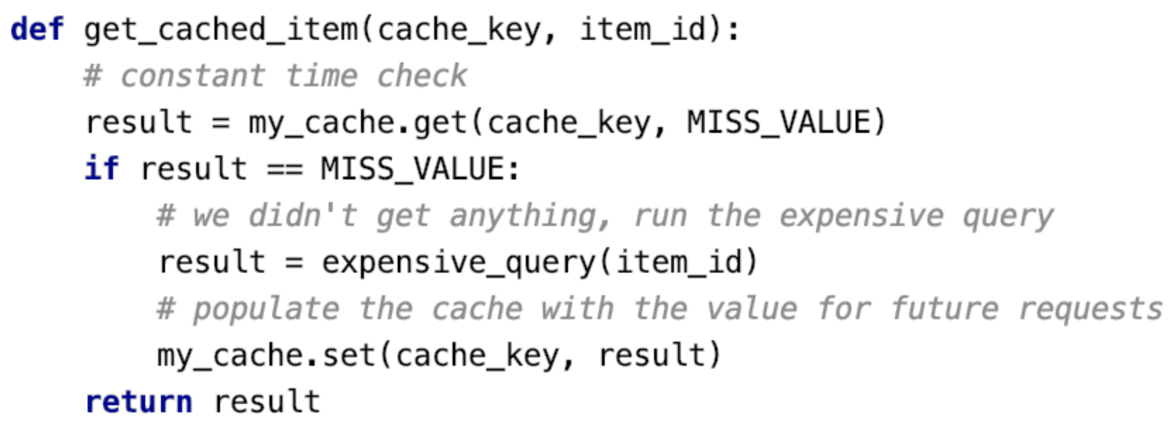
Caching

Another very common constant time optimization technique is caching. A cache is simply a key/value mapping of pre-calculated results. With caching, we ensure that we never have to perform a given calculation more than once, because the second time we can get a constant time response from the cache with the result of our value. We can shift query time into the background and ensure at-most-once computation for a given data query.

There are many different types of caches, but at Klaviyo we primarily use read-through caches, where the value is populated into the cache by the reader if it is not there, write-through caches, where the value is kept up to date by the writer whenever it changes, and asynchronously populated caches that run at regular intervals, refreshing all the relevant cache values. Caching is a great pattern to use when a memory map will not work because there is too much data, or when we have latency concerns about our application being able to perform the request in time for our use case.

A special case of cache usage is to use a non-expiring cache as a way of archiving old data in our databases. When data gets old, instead of taking up valuable space in our table indexes, the data can be retired to a low cost archival storage location. We can then collect all the metadata about this data and cache it in some permanent cache such as another database table. This type of caching needs to be permanent, so it needs to be a durable disk-backed cache rather than an in-memory cache. The reason for this is that we cannot get the data back from the database easily to recalculate the metadata once we have moved it to the archive. We can then only use the cached metadata for requests that involve that data in the future, keeping legacy information lookups fast without degrading database performance over time.

Caches in particular suffer from the staleness problem, because it can be very hard to tell which cache entries to update for a particular dataset. We can have direct caches of objects which are relatively easy to invalidate, or reset, when the object updates. However, keys that contain aggregate data are much harder to invalidate when the objects that make up those aggregates are updated. In general, unless our validation algorithms are very good, we have to assume at least a small degree of staleness in our cached data. We should also be careful about caching null result values into our cache, and should have a separate value that comes back when the key is missing rather than cached as null to distinguish between the two cases, as illustrated in the read-through cache example in figure 2.



```python
def get_cached_item(cache_key, item_id):
    # constant time check
    result = my_cache.get(cache_key, MISS_VALUE)
    if result == MISS_VALUE:
        # we didn't get anything, run the expensive query
        result = expensive_query(item_id)
        # populate the cache with the value for future requests
        my_cache.set(cache_key, result)
    return result
```

Figure 2

If we size our cache so that it has a fixed size, and hold the whole thing in memory, either in a centralized or decentralized way, we can also make sure that the data has a guaranteed freshness and does not go above the fixed size allocated for the cache. We can use key expiration to make the cache automatically dump the values under those keys after a certain time interval. This is particularly useful with read-through caches because we can have a performant constant time query per key that has a guaranteed freshness. For keeping cache size constant, we can use an eviction strategy, such as Least Recently Used or LRU, to evict keys from the cache when the cache size breaches a given threshold. The LRU algorithm is generally the best to determine which keys to evict first because most of the time we only care about the most recent keys and the most frequently accessed keys.

At Klaviyo we have a python decorator that automates the process of caching entities in a local decentralized LRU in-memory cache. The cache automatically keys the results of decorated functions to hashable function arguments, which it can then evict or expire at will. This works for making constant time operations easy, but it can be a particularly thorny problem if we require up to date data. Decentralized cache invalidation is even harder than the centralized version. If every process has its own version of the object, then updating that object will not clear the cache in all those processes, unless we set up an extremely complex publish-subscribe service that broadcasts all the keys that should be invalidated whenever something changes. In general, if we want the value to be immediately invalidated upon update, we use a centralized cache such as Memcached or a special LRU-configured Redis to make sure that our invalidation can happen immediately for everyone.

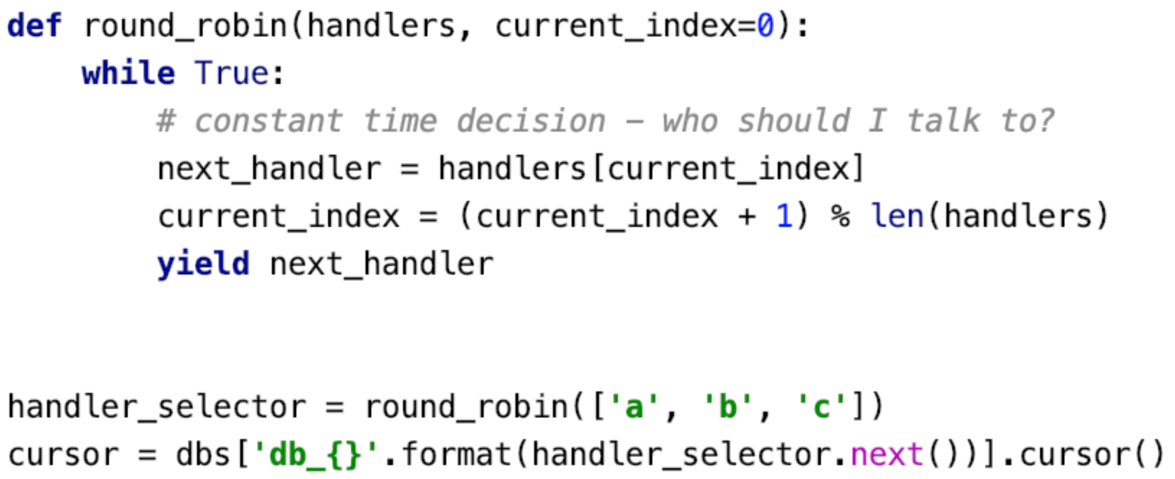
Locating Data and Computation

Besides storing and retrieving data for computation, another difficult problem is where to locate both the data and the computation. To locate our computation and therefore distribute our computation load, we can use the load balancer pattern, or a process that assigns each incoming call to a handler based on some constant time algorithm. Two common algorithms for load balancing are round robin and hash based. Round robin simply assigns each request to the next available handler, going in a circle between all the possible handlers. Round robin is considered a highly efficient algorithm in the general networking case, because

> â€œRound-robin scheduling results in max-min fairness if the data packets are equally sized, since the data flow that has waited the longest time is given scheduling priority.â€� â€” Wikipedia

The general principle applies to anything that is being measured, for example as long as each handler can compute each task in roughly the same amount of time, then splitting work in this way would also result in max-min fairness. A round robin algorithm is very easy to implement anywhere it is needed, as shown in figure 3. In special cases, we can get better performance by using the hash based algorithm, which assigns requests to the same handlers if they have a common identifier, such as a user session id.

The best time to use a load balancer algorithm for determining where to locate work is when we have a bunch of servers, brokers, or other handlers that are equally able to perform the request. We can add more handlers or remove handlers as necessary, and the load balancer will automatically adjust, pointing requests to the new set of handlers dynamically. We can store state about the requests or cache specific attributes of them if we use a hash based load balancer, because a request will likely go to the same handler every time it hits the balancer, unless the number of handlers has changed. If we want to split data rather than computation between handlers, we should not use load balancers, because if the topology changes we will not be able to find that data easily when we want to access it again.



```python
def round_robin(handlers, current_index=0):
    while True:
        # constant time decision - who should I talk to?
        next_handler = handlers[current_index]
        current_index = (current_index + 1) % len(handlers)
        yield next_handler


handler_selector = round_robin(['a', 'b', 'c'])
cursor = dbs['db_{}'.format(handler_selector.next())].cursor()
```

Figure 3

For data splitting, we use the sharding pattern. Sharding is a process that durably assigns each incoming call to a handler based on some constant time algorithm. It is similar to hash based load balancing, but the configuration is permanently stored somewhere so that everyone always agrees where each sharding key should be located. If we need to update the configuration, we have to update it everywhere at once, on all the readers and the writers of the data. Sharding is magical because it allows us to distribute data between different locations and still know in constant time where that data is located.

At Klaviyo we use horizontal sharding, which means that we split the rows of the data between different locations, as opposed to vertical sharding which splits the columns. We find it easier to work with entire rows of data in our queries and have our ORM treat a database row as an entire model, rather than having to potentially perform multiple queries for each database model instance.

One downside of using sharding is that the data loses some of its locality if we split it. That means that we can no longer use simple SQL queries to roll up all of the data, for example. Instead we need to query the data out of multiple locations and then combine it in a central location if we need to do aggregations. Another downside is that the sharding key is hard to choose and hard to change. If the sharding key is not uniformly distributed with regard to the data, it will form what are known as â€œhot shards.â€� A hot shard is when one sharding key has much higher prevalence than another, which means that data and data processing is not evenly distributed between the keys. One shard can become so hot that it might require dedicated hardware for storage and processing so that it can continue to serve requests in a timely manner.

One way to fix the hot shard problem is to use dynamic sharding. Whereas in normal static sharding everyone simply has a large in-memory map of where each shard is, and an algorithm for determining which key goes to which shard, in dynamic sharding there is a centralized node that is used as the source of truth for shard locations. This centralized node or cluster of nodes is the only place where shard locations are stored, so it is easy to modify where each shard key is, enabling us to perform asynchronous rebalancing operations to smooth out hot shards as they occur. Dynamic sharding comes with two major disadvantages however, because the data now has even worse locality, and because the process of rebalancing the shards and keeping all the shard information in memory on a centralized node is more operationally complex than storing it in static files local to the readers and writers. Worse locality means that more query types will require cross-shard queries, where the data has to be pulled from many different locations and aggregated centrally. More operational complexity means that the central node has to have high availability and two layers of information storage: one durable layer for backups and one in-memory layer for performance. Sharding is very easy to implement in code, as in figure 4, but most of its complexity comes from storing and managing the actual shard settings, managing the sharded data, and performing complex queries.

```python
# define the sharding logic
def my_shard_function(key, num_shards):
    return int(hash_function(key)) % num_shards


# constant time decision - who should I talk to?
item_shard = my_shard_function(item_id, settings.NUM_SHARDS)
# can be used to both store to and retrieve data from a
# specific destination
cursor = dbs['db_{}'.format(item_shard)].cursor()
```

Figure 4

Dealing with Race Conditions

One problem with processing a ton of data all at once is that we inevitably need to have some level of parallel computation. Once we start to access and modify the same data with multiple parallel processes, we start to run into race conditions, where two processes race and depending on which one wins the result may be different. This creates unpredictable situations, and resolving them is usually very complex. One simple constant time method for resolving race conditions is the atomic one-way gate.

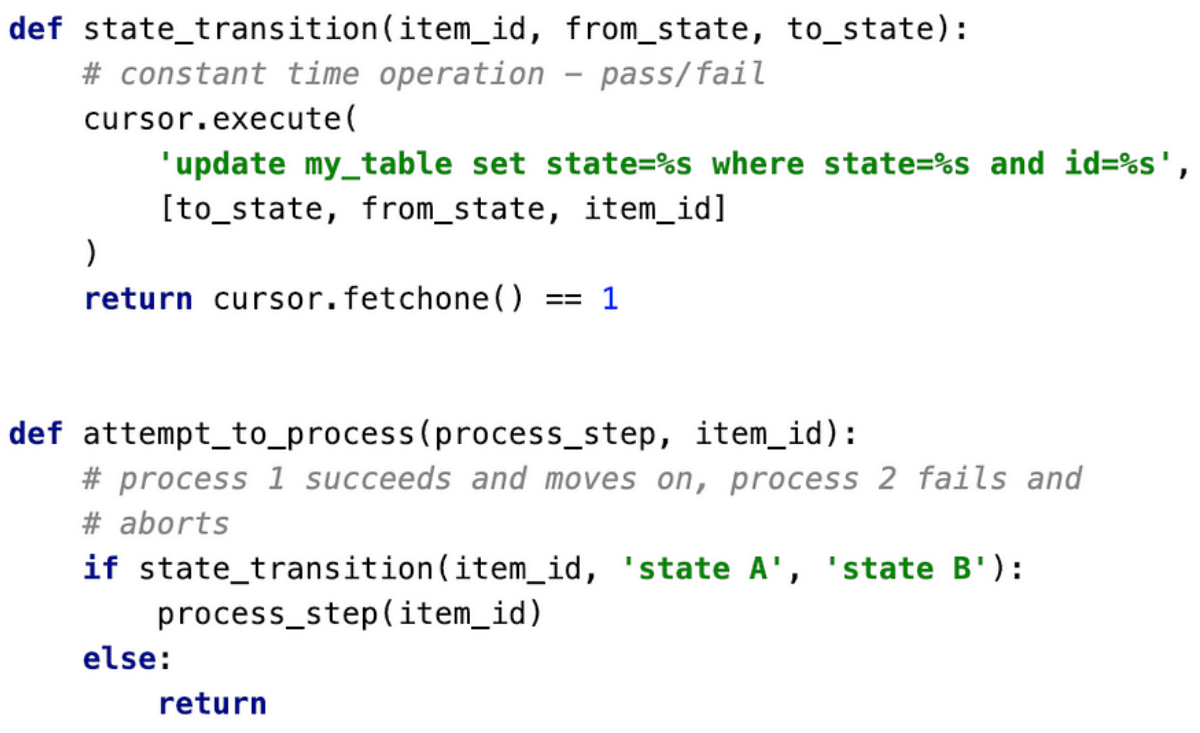
The atomic one-way gate is an at-most-once write operation that enforces a state transition. To begin with, we should define the term atomic, because it is often the source of much confusion.

> â€œAn operation acting on shared memory is atomic if it completes in a single step relative to other threads. When an atomic store is performed on a shared variable, no other thread can observe the modification half-complete.â€� â€” Preshing on programming

This means that an atomic operation can either succeed or fail when seen from the outside, and there can be no weird third option where the operation succeeds in doing something, then fails, and it is impossible to tell what actually happened. It is important for one-way gates to be atomic because they ensure that only one of many concurrent processes will pass the gate. The gate process involves an identifier, old state, and new state. The gate will only â€œpassâ€� a request for an identifier if it successfully moves from the old to the new state because of that request.

One-way gates are very useful for fast computation because they allow constant time at-most-once execution without taking out locks or building auxiliary systems. This is vital in areas where we can only do something once, such as sending a particular email to a particular customer. They can also be used to enforce a state transition to happen in only one way. For example, if our expected state flow is A->B->C->D, we cannot jump from A->C directly or go back from D->C if our one-way gate requires us to be in state B before we can be in state C. An example SQL one-way gate state transition is illustrated in figure 5.

It should be noted that a system can easily get â€œstuckâ€� in a state if it requires a one-way gate to get into that state. For example if the process dies after the A->B transition, the current state will be B but the processing for the computation between B and C has been aborted, and no one can resume the computation because no one can transition from A to B. There are several ways of dealing with this problem, but generally a synchronous or asynchronous process should detect the failure and try to reverse the gate and the operation that happened after the gate was flipped. If the operation is idempotent, simply reversing the gate and starting over will work. To minimize manual intervention, each operation in the state machine should be codified and have a standardized reversal or termination procedure in case of failure.



```
def state_transition(item_id, from_state, to_state):
    # constant time operation - pass/fail
    cursor.execute(
        'update my_table set state=%s where state=%s and id=%s',
        [to_state, from_state, item_id]
    )
    return cursor.fetchone() == 1


def attempt_to_process(process_step, item_id):
    # process 1 succeeds and moves on, process 2 fails and
    # aborts
    if state_transition(item_id, 'state A', 'state B'):
        process_step(item_id)
    else:
        return
```

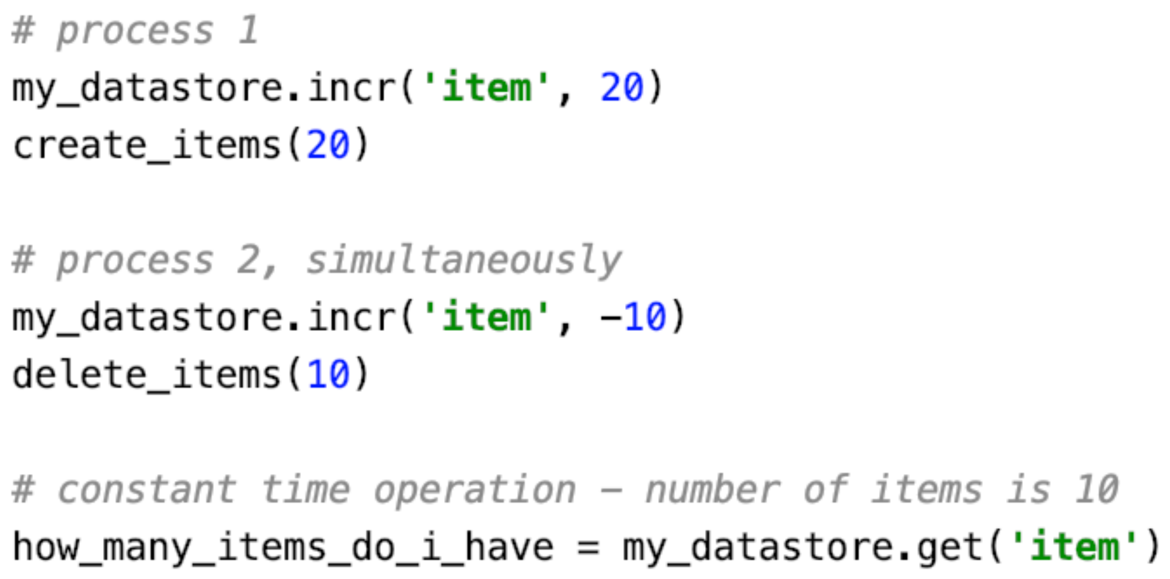
Figure 5

Counting Things

A common problem with querying a huge amount of data is that counting data points is slow and linear time. If we set up our data correctly, however, we can create conditions such that we can get accurate or mostly accurate constant time counts of almost any given dataset. Two patterns we can use for this are increments and HyperLogLogs.

Increments are processed as part of a set of commutative operations that can run in an unordered sequence on a datapoint in a centralized location. Increments make it easy to replicate data because the operations can be applied in any order to an integer and they will always end up with the same value, which relaxes many of the constraints that often have to be applied to get accurate data in distributed systems. The increment operation can be atomic, a guaranteed increment or decrement on a stored integer by a specified value â€” or failure. But even if it is not atomic, it can still be used as a cheap and mostly accurate way to offload counting into the background.

Incrementing allows us to easily solve the read-then-write race condition for integer types in databases. When an operation reads a value out of a database, modifies it, and writes it back without a lock, that creates a race condition in parallel processing. For example if two processes read the number 5 out of the database and each add 1 to that value and then write it back into the database all at once, the value will be 6 instead of the expected 7. This is because both will write 6 into the database, overwriting each other. With increments we do not have to worry about this race condition because the operation is write-only and the end value of the integer will be the same no matter who wins the writer race. The operation itself is constant time because it only changes one value in the database, and it can also be used to create constant time counts for other database entities, as in figure 6.



```
# process 1
my_datastore.incr('item', 20)
create_items(20)

# process 2, simultaneously
my_datastore.incr('item', -10)
delete_items(10)

# constant time operation - number of items is 10
how_many_items_do_i_have = my_datastore.get('item')
```

Figure 6

At Klaviyo we use increments extensively for many various processes, one of which is event aggregation. By aggregating events data as precomputed integers asynchronously we can get many metrics about events in constant time, which allows us to generate complex reports for our customers in real time.

We can also use the increment as a more advanced one-way gate if the implementation is atomic and returns the incremented value to our workers. If we define a maximum value for which the gate can â€œpassâ€�, such as only 5 operations allowed for that gate, we can attempt to increment by some number less than or equal to 5 during our one-way gate. We can know exactly what the value in the database is as a result of our increment, and subtract the previous value to find out how many values we â€œpassedâ€� for. If we get a number above 5 back, we must have â€œfailedâ€� for at least 1 operation, so we would avoid doing the operations we failed the gate for. This means that multiple processes racing for the same key would still result in only 5 operations taking place globally.

An alternative way we can count things in constant time is to use HyperLogLogs. The HyperLogLog is an algorithm for estimating the size of a dataset. HLLs allow us to calculate not only the estimated size of a given set in constant time, but also the estimated size of a union or intersection of sets. This means that unlike counters, we do not need a counter for every single possible combination of sets we want to track counts of. We can generate only one HLL per set and then dynamically combine them to answer questions such as â€œhow many elements are both in set A and set B?â€� or â€œhow many elements are in set A but not in set B?â€� in constant time.

HLLs have to be precomputed, so there has to be an asynchronous process that computes all the HLLs we will need on a regular basis, and therefore they can be stale. They are very space efficient, but in their usual configuration only accurate for large datasets.

> The math behind HLLs is complex and the theory comes from â€œthe observation that the cardinality of a multiset of uniformly distributed random numbers can be estimated by calculating the maximum number of leading zeros in the binary representation of each number in the set. If the maximum number of leading zeros observed is n, an estimate for the number of distinct elements in the set is 2^n.â€�
> â€" Wikipedia

Practically this means that we either have to have very large HLL entries in our database to estimate for small datasets, or only use small HLL entries for large sets of hundreds of thousands of members or above. We prefer to only use HLLs for large sets and to fall back to linear computation for any sets below the threshold on the assumption that the linear operation will have an upper bound in terms of execution time that allows us to think of it as constant time.

To sum up, these are some of the most impactful constant time optimization patterns that I and other Klaviyo engineers have found. These patterns have been useful for reducing the computation time of our largest workloads and keeping our services running smoothly. They are infrastructure agnostic and can be applied wherever they make sense. As data continues to grow and spread across data stores, aggregating it during user requests becomes harder and harder. Meanwhile end users and service consumers expect low latency responses and do not know or care about global data size. Therefore to create optimized and scalable code and enable these kinds of magical operations, we need to think in terms of constant time patterns to shift computation into the background and avoid live looping over data.