Designing for Database Fairness

BY IAN PYE, CO-FOUNDER & PRINCIPAL ENGINEER · DEC 21, 2015

Applying Multi-level Queues in Multi-tenant Databases

Under the hood, Kentik Detect is powered by Kentik Data Engine (KDE), a high-availability, massively-scalable, multi-tenant distributed database. One of the recurring challenges we encounter in our ongoing optimization of KDE is how best to address the need for fairness, which in computer science is another word for scheduling. What we want to achieve in practice is that queries by one customer will never cause another customer's queries to run slower than a guaranteed minimum.

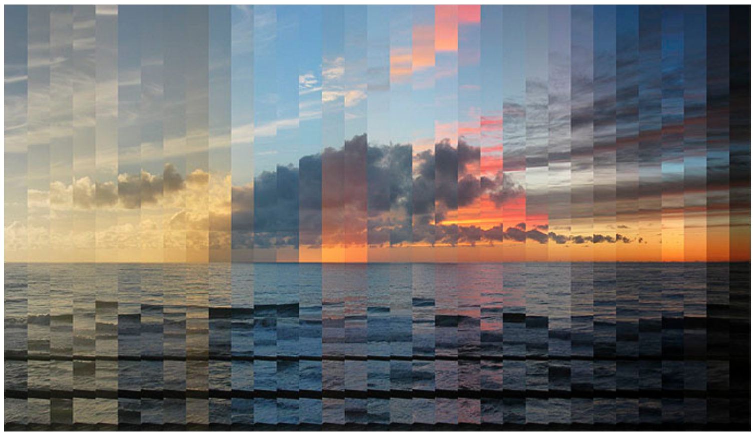
We all know that slowness is bad, but what causes it in a distributed system? Traditionally, we are primarily concerned with four main resources: network, disk, memory, and CPU. Any of these, when exhausted, will cause a system's throughput to plateau. Add more of this limited resource, and performance will increase until you hit the next resource limit. This post will cover the algorithms we use in KDE to ensure that everyone stays happy (queries don't time out) and every customer's queries get their fair share of resources.

Subquerying by time

KDE is first and foremost a time series database, which means that there is always an implicit index on event time, queries always hit this index, and the selectivity of the index (how precisely it can identify a particular record) is extremely high. Every query made by a KDE user has a time-range predicate clause, expressed as a start and end time. Dividing this continuous time range into a series of discrete subranges provides us with a logical way to break potentially vast amounts of data into chunks that can then be operated on in parallel.

To see how this plays out in practice we need to understand a little about the logical structure of KDE. As a given customer's flow records and associated data are ingested into KDE, the data from each device is stored in two parallel "main tables," one for a full resolution dataseries and another for a "fast" dataseries, which is optimized for faster execution of queries covering long timespans (see Resolution Overviewin the Kentik Knowledge Base).

These tables are each divided into logical “slices” of one minute (for Full dataseries) or one hour (for Fast dataseries). KDE handles each such slice with its own discrete subquery.



With time-based slices, the independence of subqueries holds because all of our aggregation operations — functions, like sum or min, that take multiple rows as input and return a single row — are commutative: (f(g(x)) = g(f(x))). Max of maxes and sum of sums are easy; for harder cases like mean of means, we rely on a more complex data structure to pass needed information (e.g. sample size) back to the top-level aggregation operator.

You may recall from a previous blog post, Metrics for Microservices, that the KDE system is composed of master and worker processes. Worker processes run on the machines that store “shards,” which are on-disk physical blocks that each contain one slice from a given device’s main table. Every slice is represented in KDE by two identical shards, which, for high availability, are always stored on different machines in different racks. For a given worker to handle a given subquery the shard for the timespan covered by that subquery must reside on that worker.

A given worker can only handle subqueries that cover timespans whose shards are local.

Masters, meanwhile, are responsible for splitting each query into subqueries, for identifying (via metadata lookup) which workers have access to the shards needed for a given subquery, and for assigning each subquery to a worker. The subqueries, which flow to each worker in a constant stream, are marked with two identifiers: customer_id (corresponds with customer) and request_id (corresponds with the query). The worker generates a result set from data in a shard and returns that result to the issuing master. Both the subqueries and the query as a whole have a deadline by which they must be serviced.

Workers need three local resources to service a subquery: RAM, CPU, and Disk I/O. For purposes of scheduling, we can abstract these away and just assert that for the purpose of this post the resources needed to service a subquery are represented as R.

Subquery fairness goals

To ensure fairness in the way that requests are serviced, KDE workers are designed to achieve three fairness-related goals:

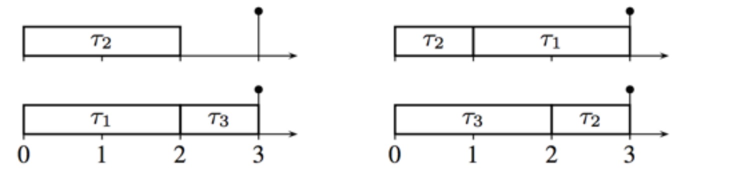
1. No customer can adversely affect another customer’s subquery response times. Adversely here means that the subquery misses its deadline. If all subquery deadlines are met, the overall query deadline will also be met.

2. No request is allowed to starve the subqueries of other requests from the same customer.
3. The system is work-conserving, meaning that if there is a subquery to run, it is run as quickly as possible, using all available resources.

Goal number 3 eliminates a static reservation system where N is the number of customers and every subquery is allocated 1/N of every resource. Instead, we are forced to adopt an elastic global system where each subquery gets from 1 to 1/M of everything, with 1/M being the minimum fraction needed to complete a subquery before a deadline (R above).

At this point you're likely thinking: "If the goal is to run every subquery before its deadline, why not run the subqueries in order of arrival? After all, first in, first out makes for a happy queue." This approach, known as EDF (earliest deadline first), turns out to be optimal in a uniprocessor system, where only one job can run at any given time. In other words, when there's just a single processor, there's no alternative scheduling algorithm that could process the workload in such a way that a deadline that would be missed under EDF would not also be missed under the alternative. For a formal mathematical proof of this, see FSU Professor Ted Baker's post on Deadline Scheduling.

EDF is nice because it presents a very simple "greedy" algorithm. KDE, however, doesn't run on a single processor; it runs on a lot of fancy servers, with R(total) > R(subquery). And that allows us to run many jobs at once. Using a simple counter, we can illustrate that EDF is non-optimal in a multiprocessor setting.



(a) τ_3 misses its deadline (b) Correct schedule with migration

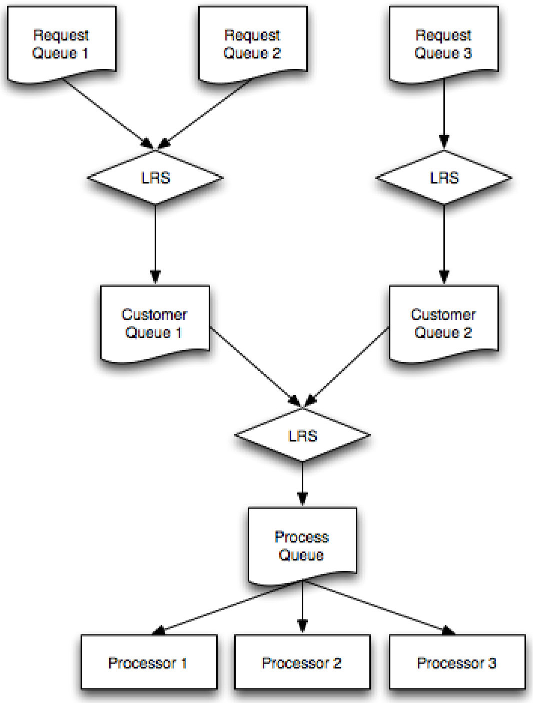
Forcing fairness with queuing

Based on the above we can see that ensuring fairness in a multiprocessor environment requires something fancier than simple EDF. What we came up with for KDE is that subqueries start in a queue of subrequests from a given request, are pulled into a queue of all subrequests for a given customer, and end up waiting for a processor in a queue with subrequests from other customers. This queue of queues of queues enables us to enforce fairness at the three points where subrequests advance from one queue to the next while still allowing us to ultimately reduce the problem to a uniprocessor model to which we can apply EDF and call it a day.

The first instance of queueing is within a request (query); each request has its own queue. A request queue is spun up when the first subrequest (corresponding to an individual subquery) is received and is halted after 600 seconds without any subrequests. This queue enforces FIFO handling of subrequests from a given query.

From here, a per-customer queue picks its subrequests from the least-recently selected (LRS) request queue that has an active subquery. Note that this request's overall deadline is earliest because at the moment it hasn't made any progress for the longest period of time. This combination ensures that all requests for a given customer make forward progress, and in essence are created equal (get equal throughput, on a weighted subrequest/sec basis).

Next the subrequests in the per-customer queues are picked for process queues, where they wait for a thread that is available to actually process an individual subquery. "P" processing threads are created, where P varies depending on the capabilities of each server (P = 1/M = R). We apply the same allocation mechanism to picking from the customer queues as we do to the request queue. Whenever a process thread is free, it notifies a central dispatch service. This dispatcher then picks the least-recently selected customer queue that has an active subquery and adds that subquery to the process queue.



The subrequests in each process queue are handled in order (EDF) by the next available process thread. All in all, this three-tiered queuing system ensures that the processors stay busy while also keeping any one customer or request from getting more than its fair share of processing power.

Master-worker scheduling

The fairness enforcement approach described above takes place within each worker. But to maximize throughput we also need to take into account how masters assign work to workers. As noted above, KDE is an HA system in which each shard is maintained on two different worker machines (if possible in two different racks). So two workers have access to the data needed for any combination of device and time-slice. The master has to decide which worker to use for each subquery to ensure fastest overall processing of a given query.

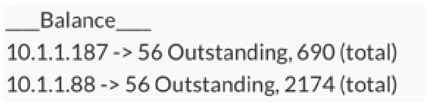
The master decides which worker for each subquery will ensure fastest processing of the overall query.

Selecting the faster worker is complicated by the fact that workers run on heterogeneous hardware. Some boxes are beefy 4U monsters and some are not. In practice, we see that some hardware combinations are three to four times more efficient than others. A query isn't finished until all of its subrequests are finished, so if half of a given query's subqueries are sent to a fast box (2x faster) and half to a slow box, the overall query time will be 2x slower than if 75% of the subqueries are directed to the fast box.

One approach to achieving the fastest aggregate throughput for all queries would be to exhaustively weight each worker so that the master can be smart about dispatching to ensure even load across workers. But in practice we've found that we can do an excellent job of keeping workers balanced by having each master keep local track of subqueries pending on each worker and prioritizing workers with the fewest outstanding subqueries.

The awesome thing about this is that every master is keeping track on its own: there's no global knowledge needed, just local knowledge. As more subqueries hit a box, the time subqueries take increases, so the chance of future subqueries going to this box goes down. We throw out all of the complications of sharing state across masters and having to statically weight workers, but we still get the real time balancing we want.

The readout at right shows an example of this balancing at work: .88 is 4x faster than .187. Both have the same number of outstanding subqueries, but .88 is handling 4x the volume. Not bad for a greedy system with zero shared knowledge.



```
___Balance___
10.1.1.187 -> 56 Outstanding, 690 (total)
10.1.1.88 -> 56 Outstanding, 2174 (total)
```

Think this is fascinating stuff? We're hiring!

Ready for more information?
Please email us at info@kentik.com or visit us at www.kentik.com.