Caching and perf. monitoring with Redis and Python

Description	An easy to use @cached decorator in Python.
Period	Spring 2016
Languages & Libs	Python
Tags	Databases, Redis

When implementing real-time APIs most of the time server load can greatly be reduced by caching frequently accessed and rarely modified data, or re-usable calculation results. Luckily Python has several features which make it easy to add new constructs and wrappers to the language, for example thanks to `*args, **kwargs` function arguments, first-class functions, decorators and so fort. Thus it doesn't take too much effort to implement a `@cached` decorator with business-specific logic on cache invalidation. Redis is the perfect fit for the job thanks to its high performance, binary-friendly key-value store with TTL and different data eviction policies and support for other data structures which make it trivial to store additional key metrics there.

There already exists many Redis libraries and even complete caching solutions, but as caching is fundamentally tricky to get right and has many edge-cases I decided to write my own (NIH :/). It also made it easy to add any nice-to-have features such as incrementing a value on the computation time histogram if a cache miss occurs. This is "sufficient statistic" to calculate total number of cache misses, average time taken, its variance and any percentiles, including the frequently used 50th percentile "median".

To-be cached functions might have some arguments which need to be skipped when calculating the cache key (such as a database connection), non-default TTL, whether to update TTL on cache hit or not and so forth. Thus `@cached` was implemented as a function which takes these configuration arguments, and returns a "wrapper" function which is the actual decorator. Python decorators are simply functions which take a single argument (the decorated function) and returns a new "wrapped" function.

There are a few conditions under which the cache code is skipped, and instead the original function's value is directly evaluated and the result is returned. One case is when Redis is not configured, for example in dev environment. An other case is when the cache key calculation fails (more on that later), of course ideally it wouldn't ever happen. A third case is when a specific "X-Skip-Redis-Cache" HTTP header is present in the API query, this is useful when running integration tests as it is important to test the full code path and not get results from the cache. It can also be used as an optimization when the client knows it is not going to benefit from caching.

As Redis is a key-value store, a "cache key" has to be generated based on the function arguments. Key is first used to check if the result already exists in Redis, and if not then it is calculated and stored there. A basic implementation has many race-conditions but in practice they should be very rare and not too critical. Function arguments can be quite large but it is better for Redis if we use relatively short keys, this is easily achieved by calculating a hash and representing it in base-64. But we can only calculate the hash from a binary representation of argument, and as they can have arbitrary types and item order may or may not be relevant (lists and tuples vs. dicts and sets) it needs to be normalized somehow. I chose encode them in JSON with alphabetically ordered keys instead of for example pickling, as it is a straightforward process to implement and understand. It doesn't handle all corner cases but gets the job done for now.

Cache invalidation is known to be difficult to get right, even more so when you have multiple versions of the software running concurrently and having been configured to different customers' different environments (dev vs. QA vs. production). Thus config file's contents and latest git commit's hash affect the generated cache key and the issue is avoided.

Results are pickled and zlib compressed to minimize memory requirements on Redis. Also on cache miss various metrics are updated such as the time it took to calculate the result, number of hits vs. misses, how large the pickled and compressed binaries were and so forth. This makes it easy to inspect these in retrospect and detect any issues on for example low hit rates, or huge objects being cached for no gain.