Optimizing Latency

By Imri Goldberg, Cymmetria VP R&D

TL;DR

- There's a lot to do beyond "standard" Python optimization
- A lot of it is around making smarter use of the DB…
- …and doing less stuff
- Code for measurement tool is available

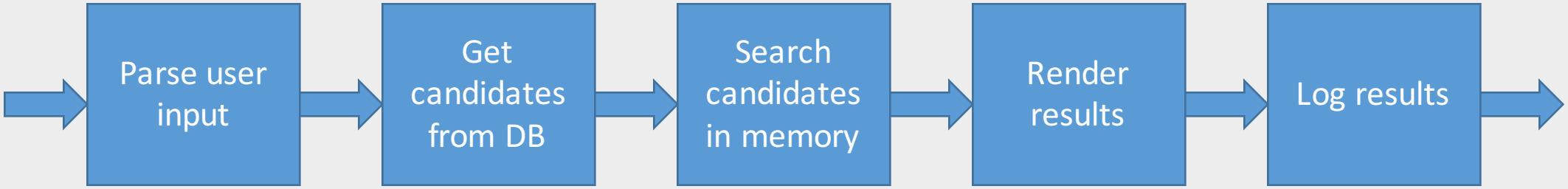
"There are only two hard things in Computer Science: cache invalidation and naming things."

-Phil Karlton

Background

- I used to be the CTO of Desti, today VP R&D of Cymmetria
- Desti was a travel search iPad app
- We made heavy use of NLP analysis of content and user input
- Search provided using our internal API, hosted on AWS
- With more content, and more complex search logic, search was becoming slow.
- REALLY slow
- Worked around by changing UX…
- …but still need to fix the underlying problem

The Problem



- Desti search was structured and ontology-based
- All steps required the ontology – parsing the input, limiting the candidate set, filtering the candidates then ranking them, then rendering the results
- It seemed everything was slow
- Every once in a while – things would be inexplicably slower
- Our stack: Apache, Python 2.7, Django, Postgresql

The problem – cont.

- I started using standard Python optimization techniques
- See http://www.algorithm.co.il/blogs/programming/pyweb-il-oresentation-on-optimization-slides/
- These weren't cutting it:
  - Profile (and cProfile) based profiling just not informative enough
  - Improvements obviously had to change architecture and general design of the system..
  - …which meant a lot of work…
  - …which means we can't afford to waste time optimizing the wrong way

Measurements

- I wanted to measure high level phases of our search
  - Specific functions and code-blocks
  - All phases of search from start to finish
  - Nested areas
- I wanted to measure real user inputs on production
  - Low Impact on current speed (also minimize observer effect)
  - Easy to turn on and off
- I needed the whole data-set: not aggregate results (except when I wanted them)

Measurement – example code

```
@code_timer.record_times
def search_main(self, user_input):
    #...
    with self._timer.record('parsing'):
        self._do_parse_phase1(user_input)
        self._do_parse_phase2()

    with self._timer.record('db-qeury'):
        query = self._prepare_query()
        candidates = self._run_query(query)
    #...
```

Measurement – example output

top_level	1.72350502	search_main	1.72350502	parsing	0.001263142	db-query	0.127518892
top_level	2.243797064	search_main	2.243797064	parsing	0.001253128	db-query	0.286036968
top_level	2.864502907	search_main	2.864502907	parsing	0.002038002	db-query	0.606631994

- No header row, as columns can change depending on flow, plan logging to avoid that.
- Nestedness depends on code flow
- You can add arbitrary data to the log, e.g. the search string, the parse result, etc.

How to use:

- Align columns correctly.
- Add average and median for each column
- Look for columns that have high average or median. These are parts of code that are consistently slow
- Look for cells that have unusually high values – these are parts of code plus inputs that result in slowness

Solutions

- Once simple optimization methods are exhausted, we need to consider methods that may require redesign or architecture changes.
- I will now list the solutions we used that were relevant to our code.

Precalculation

- Our search depended on ontology inference.
- For each possible tag and POI, we calculated the value of the tag for this POI
  - E.g. “Best Western hotel”, family friendly=0.8, “Golden Gate Bridge”, beautiful=0.9
- The original implementation had calculation happen in real time.
  - Depending on the complexity of the search-query, calculating inference could take up to **a few seconds**
- Precalculating the results of the inference and just getting them from the DB yielded significant speed gains
- The cost:
  - Offline calculation in batch required running processing “at the right time”, and losing the ability to change the ontology and see changes immediately
  - Moving slowness from compute-bound to IO-bound

Easy DB optimizations

- Everything that touched the DB was slow
- 1st Solution: minimize DB access, make the DB faster
- Now it was time to make the DB faster
- Turns out that the second run of the same query was always faster – this was because the DB already had everything in memory
- To speed up the DB:
  - Increase instance sizes, move HDs to SSD
  - Set up a read replicate. Move search queries to the read-replicate, that way the relevant tables were always in memory
- Required hiring a DBA consultant

Indexing correctly

- Some searches were still slow
- Next step was to optimize the SQL
- How to do that:
  - Dump the SQL that was run
    - If using Django ORM: str(MyObject.filter(…).query) is the easiest way, not always accurate
  - Run it on our DB console to make sure it's working
  - Run it with Explain
  - Paste the results into http://explain.depesz.com/
  - Look for places where the DB engine loops manually instead of using an index

In memory cache

- It turns out that some objects were created for each search-query
- Initial data retrieved then some calculation. Result was always the same for all calculations
- For each query that took **0.1-1 seconds**
- Easy fix: keep the result in a global variable
- Problem: how to make sure it is refreshed correctly?
  - Make it live no longer than X seconds/minutes…
  - Reset whenever source data changes
- Problem: How to reset when using multiple processes/servers?

External Caching using Elasticache

- Easy to keep short-lived, relatively small key:value pairs
- Supports versioning, so solves the problems of in-memory cache
- Our POI rendering code was slow
- Solution: cache static rendered POI data in elasticache. Since each POI is small, this is relatively easy
- Using versioning allows to keep data fresh

Logging

- After each query, we saved all results to the DB
- In order to support strong analytics, we kept everything – including rendered POIs
- Just saving the data took a lot of time!
- In django, once an HTTP response is returned, processing stops
- Solution:
  - Start a thread and do all the logging there
  - Doesn’t work well in cases of high-load, but works well enough for fixing just latency
  - For high load need to set up worker threads plus a queue

Using an indexing service

- After all optimizations took place, we still wanted to bring speed down
- The DB query had to go
- AWS has “cloudsearch” – an indexing service. It supports search by tags
- Initial proof of concept worked fast (<0.1 seconds for all queries)
- Required significant refactoring
  - Not returning too much data, as data transmission made it slow
  - Returning enough data so as not to require additional processing later
  - Cloudsearch + cache for each results = WIN

Closing words

- We managed to optimize search from multiple seconds in the worst case to always <1 second, usually faster
- The code-timer class is available https://github.com/lorg/codetimer
- Special thanks go to Nitzan Shaked, David Berlin, Tzahi Vidas and Nadav Gur

Thank you

Imri Goldberg, imri@imri.co.il