Concurrent Scraping at Scale: A Case Study

Introduction

These days, collecting information about anything and everything present in the public domain and using to find patterns that could prove crucial for various real world applications. In our use cases we had to iterate over not few, but aggregate information from tens of thousands of websites.

The factors

Following are the factors to be be considered while scraping from websites.

1. Redirects 301 and 302: Following redirects manually through code is painful.
2. Security Certificates: Certificates validation and authorisation is difficult when done through a raw HTTP call
3. Dynamic Renders: There are lots of websites which render its pages through javascript. These may not be available in the first HTTP call that is made.
4. Meta Redirects: Not just the classical redirects, the latest HTML5 specs allows or tells browsers to redirect based on Meta tags.
5. IFrames: Hitting a page through backend does not render or fetch the IFrames that the page has.
6. Forms: Following forms and submitting them becomes easier when using a real browser.

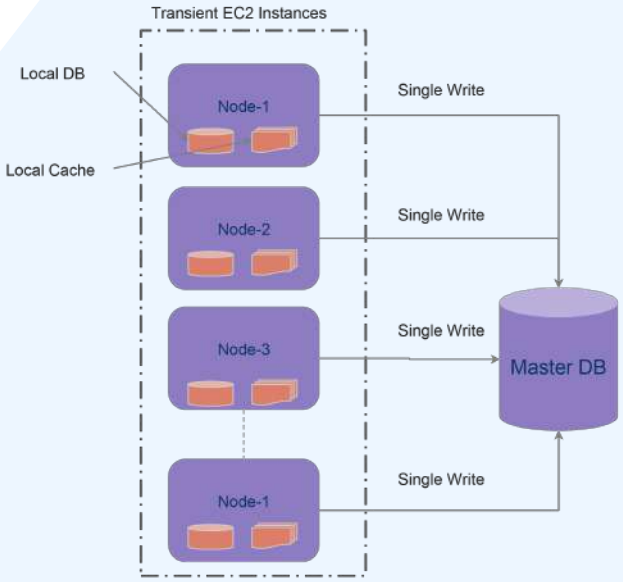
7. DDos defenders: The Website defenders like Cloudflare, incapsula, Reblaze etc make it almost impossible to crawl sites through raw HTTP calls.
8. Identification of Links: Links that are buried deep inside scripts cannot be found which means that deeper crawling will become impossible.
9. Efficiency: Since we need to crawl thousands of sites, the amount of time it would consume would really be huge. Time literally means cost here.
10. Infrastructure: We really need a large set of powerful machines in order to scrape and store the data in the scale of millions.

Pure Horizontal scalability

Considering the number of websites we had to scrape and the amount of time it would take to do that, unless we make them concurrent, it would become a near never-ending saga. So we need a vast 'pure' horizontally scalable infrastructure. In order to achieve such an architecture, we had to make sure we don't have any component in the architecture that cannot scale horizontally.

Exploring Serverless

To solve the time-consumption and infrastructure issues, we did a thorough study on going with a Serverless architecture, and to emulate a real browser, we used selenium-webdriver with chrome-extension. Lambda or the serverless engines offer good returns when we need to run things highly in parallel for tasks that could be counted in 100s of milliseconds. However, in our case, Chrome has to perform a cold start every time



Still, this was something we can live with because hitting every page via a browser solves a lot of problems like redirects, iframes, scrap defenders like Cloudflare services, better rendering of javascript forms and scripts, even meta redirects. But, because of the time it takes for cold-start of webdrivers, our cost-study had clearly revealed that we cannot continue with them anymore. We did a similar cost comparison against Amazon's EC2 infrastructure of having a dual route mechanism. Surprisingly, we found that the cost of temporary eC2 instance that simulates a lambda is comparatively lesser. The combination of transient-EC2 instances, chrome and raw crawling with selenium proved us to be the right architecture for the task to be accomplished.

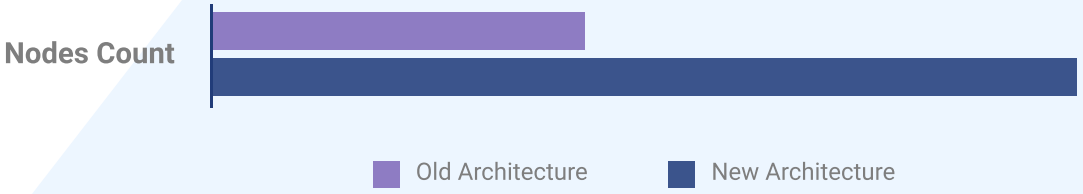
A few more of the following enhancements made this infrastructure, a fast superultra scalable architecture.

1. **Atomic Dependencies** - The interlinked components are made as atomic as possible and the interactions between them were largely minimised. Databases were localized and each atomic node will just talk to the master database once as opposed to talking thousands of times through DDL/DML queries.
2. **Caching** - Caching can be either standalone or distributed. But from atomicity perspective, standalone cache servers installed individually in our EC2 instances helped us steadfasting the process.
3. **Tracking and Error Handling** - When we trigger such a huge process, it's very vital to keep track of the progress and we should be able to spot any discrepancy and be able to take corrective action immediately. So we had installed other highly scalable error monitoring systems in place.

Conclusion

Number of AWS Nodes

For the same task, we were using some 180 t2.small nodes. For the new architecture to power boost the concurrency, we used 500 c4.large machines.



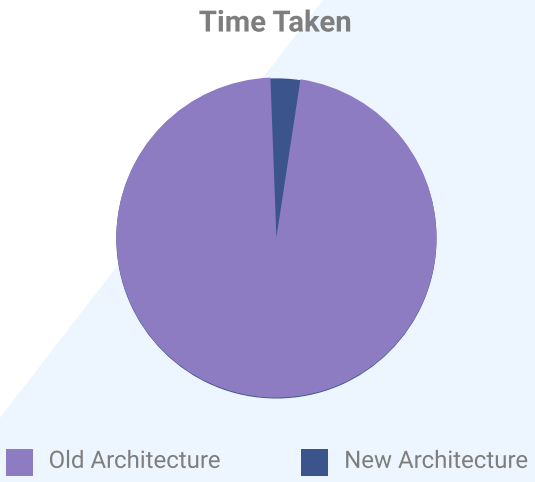
Cost Reduced

Though the number and node-config were increased, due to the least usage, the charge was drastically reduced and we were able to save more than 98 percentage.

	Number of Nodes	Node Type	Utilization	Total Cost
Old	180	t2.small	100% for 5 weeks	$ 3031
New	500	c4.large	100% for just 30 Minutes	$ 50

Time Saved

But more than the cost, the time it saved us is phenomenal! (167900%) and all the agony of waiting for whole 5 weeks to get this done is totally gone away.



Francium Tech's approach to this problem is what differentiates us with others. Our focus was more on achieving efficiency on all levels. As one could notice, we not only reduced the cost by multi-folds, but also the turn-around time taken to handle any foreseen inconsistencies the system threw. Earlier it took weeks, but, with this new architecture it was reduced to minutes. This directly correlates the stakeholder's trust on the stability of the entire machine. In other words, the ball was hit out of the park.