

Nutch2Crawling

This document describes the crawling jobs of Nutch2 (NutchGora).

- [Introduction](#)
- [Generate](#)
 - [Mapper](#)
 - [Partitioning](#)
 - [Reducer](#)
 - [Result](#)
 - [Things for future development](#)
 - [Is it really necessary to \(re\)process every url each generate?](#)
 - [Suggestions](#)
- [Fetch](#)
 - [Mapper](#)
 - [Partition](#)
 - [Reduce](#)
 - [Graphic Overview](#)
- [Parse](#)
 - [Mapper](#)
- [DbUpdate](#)
 - [Mapper](#)
 - [Partition](#)
 - [Reduce](#)

Introduction

A crawl cycle consists of 4 steps, each implemented as an Hadoop job.

- [GeneratorJob](#)
- [FetcherJob](#)
- [ParserJob](#) (optionally done during fetch using 'fetcher.parse')
- [DbUpdaterJob](#)

To populate initial rows for the webtable you can use the [InjectorJob](#).

There is a single table **webpage** that is the input and output for these jobs. Every row in this table is an url (WebPage). To group urls from the same TLD and domain closely together, the row key is stored as url with **reversed host components**. This takes advantage of the fact that row keys are sorted (in most NoSQL stores). Scanning over a subset is generally a lot faster than scanning over the entire table with specific rowkey filtering. See the following example rowkey listing:

- com.example.www:http/
- com.example.www:http/about/
- net.sourceforge:http/
- ...
- org.apache.gora:http/
- org.apache.nutch:http/
- org.apache.www:http/
- org.wikipedia.en:http/
- org.wikipedia.nl:http/

Generate

Creates a new batch. Selects urls to fetch from the webtable (or: marks urls in the webtable which need to be fetched).

Mapper

Reads every url from the webtable.

1. Skip the url if it has been generated before (has a generated mark (batch ID)). This will allow you to run multiple generates. 2. (optional) Normalize the url (apply URLNormalizers) and filter the url (apply URLFilters). Disable for fast generating of urls. 3. Is it time to fetch this url? (fetchTime < now) 4. Calculate scoring for this url. 5. Outputs every url that needs to be fetched, together with its score (SelectorEntry).

Partitioning

All urls are partitioned by domain, host or IP (the partition.url.mode property). This means that all urls from the same domain (host, IP) end up in the same partition and will be handled by the same reduce task. Within each partition all urls are sorted by score (best first). Notes:

- (Optional) Normalize urls during partitioning. Disable for fast partitioning of urls.
- When partitioning by IP: this might be heavy on DNS resolving!

Reducer

Reads every url from the partition and keeps selecting urls until the total number of urls has reached a limit, or the number of urls per domain (host, IP) has reached a limit.

1. Stop including urls when we have reached topN/reducers urls. This will give topN urls for all reducers together. 2. Stop including urls for a certain domain (host, IP) if the number of urls for that domain (host, IP) exceeds generate.max.count. The reducer keeps track of this using a map. This works because all urls from the same domain (host, IP) are handled within the same reduce task. Note that if the number of different domains (hosts, IPs) per reducer is large, the map may become large. However, one can always increase the number of reducers. 3. For each selected URL, write a generate mark (batch ID). 4. Output the row to the webtable.

Result

A maximum of topN urls gets selected. That is: these get a generator mark in the webtable. However, there are two reasons why not always the topN **best** scoring urls are selected. Or not even topN urls at all (while there are enough urls to fetch):

1. The property generate.max.count limits the number urls per domain (host, IP). So urls from domain abc.com may be skipped (we already have enough urls from abc.com) while urls with lower score from xyz.com still can get in (still room for more urls from xyz.com). This is good: we rather have lower scored urls than too much from the same domain (host, IP). 2. Each reducer (partition) may generate up to topN/reducers urls. But due to partitioning not all reducers may be able to select this amount. This is an implementation consequence. However, if number of domains (hosts, IPs) is much bigger then the number of reducers then this is not really an issue. And, we get scalability in return.

Example:

Settings:

- topN = 2500
- generate.max.count = 100
- reducers = 5

Input for partition 1:

- abc.com: 10 urls
- klm.com: 100 urls
- xyz.com: 1000 urls

Output from partition 1:

- abc.com: 10 urls
- klm.com: 100 urls
- xyz.com: 100 urls

So for this partition, only 210 urls will be selected for fetching.

Things for future development

How does this work together with fetching? Why write the generator mark back to the webtable and then select everything again, do the same partitioning and start fetching. Can't we do the fetching in the Generator reducer directly? Possible cons:

- If the job fails, you need to do the generate from scratch.

Pros:


- No need for another job (select, partition...)

Is it really necessary to (re)process every url each generate?

It sounds very inefficient to read all urls from the webtable for each generate job again, perform the checks above (calculate scoring), do sorting and partitioning...Especially when the number of urls >> topN. Sure, after fetching a batch, the webtable has changed (new urls, scoring changed) so if you really want to do things perfectly, you should check the webtable again completely... right...?

- In production it seems generating is not that expensive because there is limited data on the input. (Just the fields necessary for determining the score and whether it should be fetched).

Suggestions

- Use region side filtering (where possible) to check the generated mark and fetchtime.
- If scoring isn't important : keep track of url limits (topN and per domain (host, IP)) in the mapper. And stop reading new input records when topN /mappers have been selected. For domain and host mode (generate.max.count) this actually may work relatively fast since the map input is sorted by inverse url.

Cons:

- Scoring is not used for selection
 - Domains (hosts) at the start of a region (mapper input) have the highest chance to get selected.
-

Fetch

Fetches all urls which have been marked by the generator with a given batch ID (or optionally fetch all urls)

Mapper

Reads every url from the webtable.

1. Check if the generator mark matches. 2. If the url also has a fetcher mark, the skip it if the 'continue' argument is provided, else also refetch everything that has been fetched before in this batch. 3. Output each url with a random int as key (shuffle all urls)

Note: we need to go through the whole webtable here again! Possible changes:

- Use region side filters to check generator marks?
- Let the generator reducer do the fetching? Cons:
 - No random ordering of selected urls
- Put generator output in a intermediate location (file/table) and fetch from there (back to segments in Nutch 1.x)

Partition

Partition by host.

Note: Why does the fetcher mapper use its own partitioner (by host)? The fetcher reducer supports queues per host, domain and IP...

Reduce

1. Puts the randomized urls in fetch queues (one queue per domain (host, IP) for politeness) and scans the queues for items to fetch. 2. Fetch each url. Todo: How are redirects handled? 3. Output success and failures to the webtable. Note: fetchTime is set to 'now'. 4. If parsing is configured: parse the content (skipping unfetched items) using the [ParseUtil](#) class (also used by the [ParseJob](#)).

Graphic Overview

[Nutch_2.x_Fetch_Architecture](#)]]

Parse

Parses all webpages from a given batch id.

Mapper

1. Reads every webpage from the webtable 2. Check if the fetch mark matches. 3. If this webpage has been parsed before (even in another batch) then skip it. 4. Run parsers on the row. 5. Output the row to the webtable.

DbUpdate

Updates all rows with inlinks (backlinks), fetchtime and the correct score.

Mapper

1. Read every row from the webtable. 2. Update the score for every outlink. 3. Output every outlink with score and anchor (linktext). 4. Output the rowkey itself with score.

Partition

Partition by {url}. Sort by {url,score}. Group by {url}. This ensures the inlinks are sorted by score in the reducer.

Reduce

1. The key in the reducer is a row in the webtable and the reduce values are it's inlinks. 2. Update the fetchtime. 3. Update the inlinks (capped by the property 'db.update.max.inlinks'). 4. Update the score for a row based on it's inlinks. 5. Output the row to the webtable.