Apache Solr Popularity Technical Details

Intended audience:

- Developers looking for low-level details in order to make modifications or improvements to the popularity algorithm.

Scientists in information retrieval that are interested in collaborating on a paper for peer review are encouraged to contact me via a private message (contact info at bottom). This work is novel research on increasing the performance of enterprise search engines.

Preliminaries

It is recommended to be familiar with the Apache Solr Popularity Configurations document prior to reading this paper, as it will assume you are familiar with terms discussed there. A link to the document can be obtained in the Document Section of the Apache Solr Popularity project page.

Intro

The purpose of this project is to incorporate the popularity of a node into the search results. The importance of this can be illustrated by a simple example. Say a user is searching on a university website for a webpage on a particular biology course. After submitting the search query, Solr returns two pages in the search results, with Page A having a score of 0.74 and Page B having a score of 0.72. Without additional information, Solr will place Page A first, followed by Page B. For illustration purposes, say Page A has a monthly hit count (page views) of 7 and Page B has a monthly hit count of 547. Considering the scores are very similar (Page A: 0.74 vs. Page B: 0.72), yet Page B is in much greater demand, it more likely that user is more interested in Page B instead of Page A. Therefore, Page B should be ranked higher than Page A; however, this is not what happens.

This simple example happens all too often when we search for a specific page on websites, particularly private ones, and our search query returns search results listing webpages that few people would be interested in. Public search engines can use algorithms such as page rank to remove less useful pages; however, algorithms such as page rank are much less useful for enterprise search engines. To make up for this deficient, the popularity of a node can be used. Unfortunately, the Apache Solr Search Integration module does not collect popularity information for Solr, and thus, Solr cannot use this highly valuable information to rank the nodes more appropriately. This is the problem that the Apache Solr Popularity module addresses and this paper's purpose is to detail the algorithms used to perform this function effectively.

Before we can use popularity to re-rank the pages more effectively, a model of popularity must be derived, as detailed in the Popularity Model section. Next, popularity must be transformed into a form that can be used effectively for ranking; this is detailed in the Ranking Modifier section. Finally, the Popularity Algorithm section discusses the algorithm developed.

Popularity Model

At the most fundamental level, popularity is based on the hit count for each node. Thus, it is necessary to track the hit counts c_t^n for each node n at time t and calculate the respective popularity P_t^n at each time interval Δt (e.g., at each cron update). The simplest method of calculating the popularity is to set the popularity P_t^n to the hit count c_t^n (i.e. $P_t^n = c_t^n$). A major problem with this method is that it does not take into consideration the amount of time t the node has been tracked for. A node may have a high hit count simply because it has been tracked for a longer time and not necessarily because it is popular. To remedy this, the frequency of the hit count can be used, which provides an alternative definition for the popularity (i.e. $P_t^n = f_t^n$).

The frequency can vary greatly from one website to another. In order to use values that are more consistent between websites, the frequency f_t^n can be normalized by the frequency of the node with the greatest frequency $f_t^{\max}$, which scales the node popularity to between 0 and 1. To improve user readability, it is multiplied by 100 to rescale it from 0 to 100. Formally, the popularity $P_{\Delta t}^n$ over the time interval Δt can be defined as

$$P_{\Delta t}^n \stackrel{\text{def}}{=} \left(100 \cdot \frac{f_t^n}{f_t^{max}}\right) \tag{1}$$

Factoring out the time component from the frequency gives

$$P_{\Delta t}^n \stackrel{\text{def}}{=} \left(100 \cdot \frac{f_t^n}{f_t^{max}}\right) = \left(100 \cdot \frac{c_t^n \cdot \Delta t}{c_t^{max} \cdot \Delta t}\right) \tag{2}$$

which can be reduced to a normalized version of the hit count

$$P_{\Delta t}^n \stackrel{\text{def}}{=} \left(100 \cdot \frac{c_t^n}{c_t^{max}}\right) \tag{3}$$

Eq. (3) provides that base for our calculations of popularity.

Next, at each popularity update, the values may vary greatly, simply because of the partially random nature of user browsing behavior. To remove this random noise, it is useful to perform some form of filtering (e.g., averaging). The traditional method to calculate an average is to sum up all values and divide by the total number of values (i.e. batch average). However, this is problematic for modeling popularity because

the time interval Δt between popularity updates may not be constant. Therefore, it is necessary to perform a weighted average based on the size of the time interval Δt.

Unfortunately, calculating batch averages at each update is both computationally inefficient and inefficient on memory. Alternatively, an iterative average can be used. The iterative average has the advantage that rather than having to keep track of every value calculated, only the previous average is required, which is beneficial since it greatly reduces the number of calculations at each update and greatly reduces memory requirements since recording the complete history is not required.

In the most basic form, the iterative average of each node as it applies to popularity can be calculated by

$$P_t^n = \left(\frac{\Delta T}{T_t^n}\right) P_{\Delta t}^n + \left(\frac{T_t^n - \Delta T}{T_t^n}\right) P_{t-1}^n \tag{4}$$

where P_t is the popularity at the current time t, P_{t-1} is the popularity at the previous update, and $P_{\Delta t}$ is the popularity since the last update. Furthermore, T_t^n is the current (total) time that the node n has been tracked, and ΔT is the time interval since last update. Specifically, the relation between the current time T_t^n, the previous time T_{t-1}^n, and the time interval ΔT is defined by

$$T_t^n \stackrel{\text{def}}{=} T_{t-1}^n + \Delta T \tag{5}$$

The problem with this method of calculating popularity using Eq. (4) is that if the node was in demand in the past but is no longer in demand now, it may be calculated as being popular despite being currently viewed infrequently. Inversely, if a node had low demand in the past, but changes were made to make it more in demand, it may still be calculated as unpopular. To avoid this, it is beneficial to have the past popularity values decay, which will cause newer popularity calculations to hold more weight.

One solution to allow the past to decay is to have the time based on function L of the previous time, which compresses (reduces) larger times. The first step is to redefine time from Eq. (5) to being a function of the previous time

$$T_t^n \stackrel{\text{def}}{=} L(t_t^n) = L(t_{t-1}^n + \Delta t) \tag{6}$$

and

$$\Delta T \stackrel{\text{def}}{=} L(\Delta t) \tag{7}$$

where t_t is the (unadjusted) time at the current time step, t_{t-1} is the (unadjusted) time at the previous time step, and Δt is the (unadjusted) time step interval.

Substituting Eq. (6) and Eq. (7) into Eq. (4) provides the popularity P_t^n for node n

$$P_t^n = \left(\frac{L(\Delta t)}{L(t_{t-1}^n + \Delta t)}\right) P_{\Delta t}^n + \left(\frac{L(t_{t-1}^n + \Delta t) - L(\Delta t)}{L(t_{t-1}^n + \Delta t)}\right) P_{t-1}^n \tag{8}$$

Starting with the following identity and rearranging

$$P_{t-1}^{n} = P_{t-1}^{n} \tag{9}$$

$$P_{t-1}^{n} = \left(\frac{L(t_{t-1}^{n} + \Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{t-1}^{n} \tag{10}$$

$$P_{t-1}^{n} = \left(\frac{L(t_{t-1}^{n} + \Delta t) + \big(L(\Delta t) - L(\Delta t)\big)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{t-1}^{n} \tag{11}$$

$$P_{t-1}^{n} = \left(\frac{L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{t-1}^{n} + \left(\frac{L(t_{t-1}^{n} + \Delta t) - L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{t-1}^{n} \tag{12}$$

Eq. (9) can be put in the form

$$\left(\frac{L(t_{t-1}^{n} + \Delta t) - L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{t-1}^{n} = P_{t-1}^{n} - \left(\frac{L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{t-1}^{n} \tag{13}$$

Substituting Eq. (13) into Eq. (8)

$$P_{t}^{n} = \left(\frac{L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{\Delta t}^{n} + P_{t-1}^{n} - \left(\frac{L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right) P_{t-1}^{n} \tag{14}$$

and rearranging provides the popularity update function

$$P_{t}^{n} = P_{t-1}^{n} + \left[\frac{L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right] [P_{\Delta t}^{n} - P_{t-1}^{n}] \tag{15}$$

As clarified by the following, Eq. (15) can be seen as a mathematical filter

$$P_{t}^{n} = P_{t-1}^{n} + \underbrace{\overbrace{\left[\frac{L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right]}^{K} \cdot \overbrace{\left[P_{\Delta t}^{n} - P_{t-1}^{n}\right]}^{residual}}_{correction} \tag{16}$$

where the new popularity P_t^n is obtained iteratively by taking the previous popularity P_{t-1}^n and adding a correction to it. The correction is calculated by multiplying a gain function K by the residual, where the residual is the difference between the old popularity P_{t-1}^n and the recent

popularity $P_{\Delta t}^{n}$, and the gain function K determines how much of the residual to include into the estimated new popularity P_{t}^{n}. The K has a value between 0 and 1.

The constraint for the gain function K is that it outputs a value between 0 and 1 and the constraint for the decay function L is that when the values are small, the output equals the input (i.e. $\lim_{t\to 0}\big(L(t)\big) = \lim_{t\to 0}(t)$)

The simplest decay function L that can be used would be a simple linear function (i.e. $L(t) = t$). However, this does not allow older popularities to decay. Alternatively, if we compress (reduce) larger values of t, yet do not compress smaller values of t, then it will have the same effect as the past decaying, which results in newer popularity values holding more weight. An effective way to model this effect is to use a variant of the logistic function

$$L(t) = \left(\frac{2}{1 + e^{\left(-2\frac{t}{T_w}\right)}} - 1\right) \cdot T_w \tag{17}$$

where $\mathrm{T_w}$ is the time window. As t approaches $\mathrm{T_w}$, then $L(t \to \mathrm{T_w})$ becomes close to 1. However, only as t gets very large does it equal 1 (i.e. $\lim_{t\to\infty} L(t) = 1$).

Rearranging Eq. (17) gives

$$L(t) = \left(\frac{1 - e^{\left(-2\frac{t}{T_w}\right)}}{1 + e^{\left(-2\frac{t}{T_w}\right)}}\right) \cdot T_w \tag{18}$$

By isolating the gain function K from Eq. (16)

$$K = \left[\frac{L(\Delta t)}{L(t_{t-1}^{n} + \Delta t)}\right] \tag{19}$$

we can substitute Eq. (18) into Eq. (19) to obtain the logistic gain K

$$K = \left[\left(\frac{1 - e^{\left(-2\frac{\Delta t}{T_w}\right)}}{1 + e^{\left(-2\frac{\Delta t}{T_w}\right)}}\right) \cdot \left(\frac{1 + e^{\left(-2\frac{t_t^n}{T_w}\right)}}{1 - e^{\left(-2\frac{t_t^n}{T_w}\right)}}\right)\right] \tag{20}$$

Substituting Eq. (20) back into the update function $P_t^n = P_{t-1}^n + K \cdot [P_{\Delta t}^n - P_{t-1}^n]$, we obtain the complete popularity update function

$$P_t^n = P_{t-1}^n + \overbrace{\left[\left(\frac{1 - e^{\left(-2\frac{\Delta t}{T_w}\right)}}{1 + e^{\left(-2\frac{\Delta t}{T_w}\right)}}\right) \cdot \left(\frac{1 + e^{\left(-2\frac{t_t^n}{T_w}\right)}}{1 - e^{\left(-2\frac{t_t^n}{T_w}\right)}}\right)\right]}^{K="logistic\ gain"} \cdot [P_{\Delta t}^n - P_{t-1}^n] \tag{21}$$

Eq. (21) allows the popularity to be calculated iteratively, weighs the influence of the update proportional to the amount of time that has passed, smooths out noise, and decays the past to positively bias more recent calculations. However, it does not weigh times with high server load greater than times of low server load.

There is an advantage to weighing periods of high total hit counts greater than periods of low total hit counts because node popularity calculations during the former will more closely resemble the individual demands for the node by more users than the latter. In order to bias times of high total hit count times, the time step interval since the last update can be redefined as the count-adjusted time interval. Specifically,

$$\Delta t \stackrel{\text{def}}{=} \frac{f_t}{\bar{f}_t} \tag{22}$$

where f_t is the frequency of hit counts for all nodes since the last update, and $\bar{f}_t$ is the average daily frequency of hit counts for all nodes. By factoring out the time component Eq. (22) becomes

$$\Delta t \stackrel{\text{def}}{=} \frac{c_t}{\bar{c}_t} \tag{23}$$

where c_t is the total number of hit counts for all nodes since the last update, and $\bar{c}_t$ is the average total hit counts for all nodes. To prevent all of the equations from being unnecessarily redefined, for simplicity, the count-adjusted time interval will be denoted by Δt and the true time interval will be denoted by $\Delta\tau$.

The average hit count $\bar{c}_t$ can be calculated using a similar filter as used for calculating popularity

$$\bar{c}_t = \bar{c}_{t-1} + K \cdot [c_t - \bar{c}_{t-1}] \tag{24}$$

which can be expanded into

$$\bar{c}_t = \bar{c}_{t-1} + \left[\left(\frac{1 - e^{\left(-2\frac{\Delta t}{T_c}\right)}}{1 + e^{\left(-2\frac{\Delta t}{T_c}\right)}}\right) \cdot \left(\frac{1 + e^{\left(-2\frac{t_t}{T_c}\right)}}{1 - e^{\left(-2\frac{t_t}{T_c}\right)}}\right)\right] \cdot [c_t - \bar{c}_{t-1}] \tag{25}$$

where t_t is the total time the module has been enabled for, and the constant T_c is defined by

$$T_c = max(T_w, T_m) \tag{26}$$

where $\mathrm{T_m}$ is a value greater than 1 (day) to prevent $\mathrm{T_c}$ from dropping too low. This ensures that the average rate remains accurate and is not overly influenced by sub-daily fluctuations in hit counts.

Note: when updating the time at the end of each update, we use the true time τ_t^n as calculated by

$$\tau_t^n = \Delta\tau + t_{t-1}^n \tag{27}$$

Furthermore, at the start of each update, the time step is moved forward

$$t_{t-1}^n = \tau_t^n \tag{28}$$

However, the trivial step of Eq. (28) is solely presented for formal accuracy.

Ranking Modifier Model

Popularity provides a good indication of the demand for a node; however, it is not directly useful as a variable to modify Apache Solr's ranking scores. To do this, we define a new variable, the ranking modifier r_t^n for a node n, which is multiplied against Solr's ranking scores s_q^n for query q to create the popularity modified ranking score $\hat{s}_q^n$. Formally, the popularity modified ranking score is obtained by

$$\hat{s}_q^n = s_t^n \cdot r_q^n \tag{29}$$

The modified ranking score $\hat{s}_q^n$ from Eq. (29) is what Solr uses to rank the nodes from a search query. Since it is used for ranking only, the specific absolute values are not important, but rather, it is the relative values that determine the order.

The ranking modifier r_t^n is based off the popularity value and is molded into a form that will be useful for search. Specifically, the ranking modifier r_t^n is defined as

$$r_t^n \stackrel{\text{def}}{=} \gamma + \alpha \cdot N\left(P_t^n, P_t^{max}, P_t^{min}\right) \tag{30}$$

where α is the popularity influence coefficient, γ is the low-popularity influence constant, N is the normalization function, $P_t^{\max}$ is the popularity of the node that currently has the largest popularity, and $P_t^{\min}$ is the popularity of the node that currently has the lowest popularity.

The popularity influence coefficient α is the amount that node popularity scores influence the ranking modifier. The low-popularity influence constant γ limits the decrease in the value of the ranking modifier for nodes with a low popularity, which prevents unpopular nodes from being difficult to find in the search results. The normalization function N scales and normalizes the popularity values, and determines how much the popularity of highly popular nodes is compressed (reduced in value). This prevents very popular nodes from dominating the search results.

There are three types of normalization/compression functions: linear, square root, and logarithmic. Linear normalization is defined as

$$N_{LIN}\left(P_t^n, P_t^{max}, P_t^{min}\right) = \left(\frac{P_t^n - P_t^{min}}{P_t^{max} - P_t^{min}}\right) \tag{31}$$

and provides no compression, which does not reduce the influence of overly popular nodes. Square root normalization is defined as

$$N_{SQRT}\left(P_t^n, P_t^{max}, P_t^{min}\right) = \left(\frac{P_t^n - P_t^{min}}{P_t^{max} - P_t^{min}}\right)^{\frac{1}{2}} \tag{32}$$

and provides a moderate amount of compression, which partially reduces the influence of overly popular nodes. Finally, the logarithmic normalization is defined as

$$N_{LOG}\left(P_t^n, P_t^{max}, P_t^{min}\right) = \frac{log\left(1 + P_t^n - P_t^{min}\right)}{log\left(1 + P_t^{max} - P_t^{min}\right)} \tag{33}$$

and provides a high degree of compression, which causes a greater reduction of influence of overly popular nodes.

Popularity Algorithm

Using the defined equations, an algorithm can now be detailed. Page views are continuously recorded, and at each cron update, the following is executed:

1. Calculate the recent popularity $P_{\Delta t}^n$ using Eq. (3)
2. Calculate the average hit count $\bar{c}_t$ using Eq. (25) and Eq. (26)
3. Calculate the count-adjusted time interval Δt using Eq. (23)
4. Calculate the count-adjusted current time $t_t^n = t_{t-1}^n + \Delta t$
5. Calculate the new popularity P_t^n using Eq. (21)
6. Calculate N using either Eq. (31), Eq. (32), or Eq. (33)
7. Calculate the ranking modifier r_t^n using Eq. (30)
8. Calculate the true current time τ_t^n using Eq. (27)
9. Store and update the true current time τ_t^n, the new popularity P_t^n, and the ranking modifier r_t^n in the database
10. Repeat Steps 1-9 for the remaining nodes
11. Send ranking modifiers to Solr and reload Solr's cache

Using Steps 1-11, the popularity and ranking modifiers are calculated and used to re-rank nodes by including the demand for a node to be included into Solr's ranking.

When properly tuned, this module can greatly improve the relevancy of a search.

Credits

Developer:

Jonathan Gagne (jongagne)
http://drupal.org/user/2409764

This project was funded by:

OPIN Software
http://www.opin.ca/