A Refreshing Perspective of Search Engine Caching

Cambazoglu, B.B.; Junqueira, F.P.; Plachouras, V.; Banachowski, S.;
Cui, B.; Lim, S.; Bridge, B.

(WWW 2010)

High Performance Computing Laboratory
Istituto di Scienza e Tecnologie dell'Informazione (ISTI)
Consiglio Nazionale delle Ricerche (CNR)
Pisa, Italy
diego.ceccarelli@isti.cnr.it

May 28, 2010

Outline

The problem

- **Achieve low latency**: large result caches
- **Problem**: cache entries may become stale
- **Freshness**
- **Not eviction policies**, but the ability to cope with changes to the index

They propose:

- A novel algorithm to set cache entries to expire
- Heuristic that combines the **frequency** of access with the **age** of an entry in the cache
- **Refresh rate**: mechanism that takes into account idle cycles of back-end servers
- **Results**: using a real workload, the algorithm can achieve hit rate improvements as well as reduction in average hit ages

Result Caches

- Crucial performance components
  - to reduce the query traffic to back-end servers
  - to reduce the average query processing latency
- Query frequencies follow a power-law distribution $\Rightarrow$ result cache implementations in practice can achieve high hit rates
- The problem is not memory space! (time to process a query on a search cluster $\approx$ time to fetching from disk)

Freshness

- Significant fraction of previously computed results in the cache become stale over time

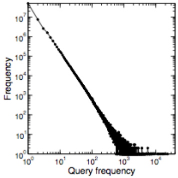
- The freshness problem becomes more severe as the cache capacity increases

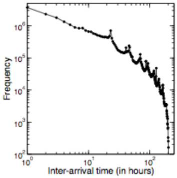
Solutions

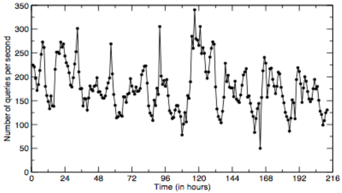
- Invalidate entries over time
- Two possible approaches for cache invalidation in this context:
  1. **coupled** difficult to realize in practice
  2. **decoupled** using a **time-to-live (TTL)** value
- The engine is often not processing queries at full capacity
- They can exploit idle cycles to re-process queries and *refresh* cache entries

Query Log

- Query log obtained from the traffic of the Yahoo! Web search engine
- 130,320,176 queries
- 65,100,647 unique
- Nine consecutive days of operations
- 49,679,763 singleton queries
- Most frequent query appears 372,447 times

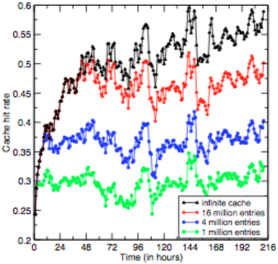


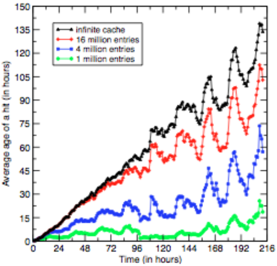




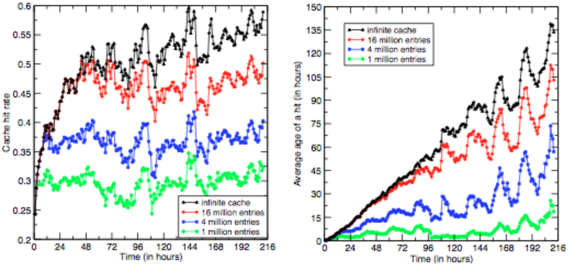
Cache Capacity

- Hit-rate increases as they keep more entries in the cache





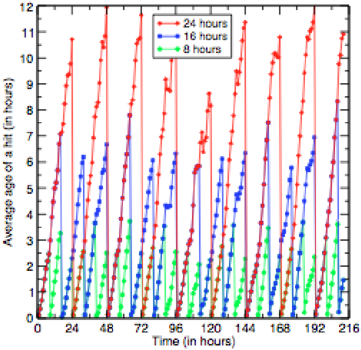
- Practical result caches in large WSE perform approximately as infinite cache
- **Problem!** the freshness of results determines the quality perceived by users
- Freshness is an issue even for small caches!



- **Average age of a hit**: $t_{hit} - t_{update}$
- Freshness problem is more severe with the infinite cache
- After nine days, the average age of a hit on the infinite cache becomes about 5.6 days
- The following experiments assume an infinite cache

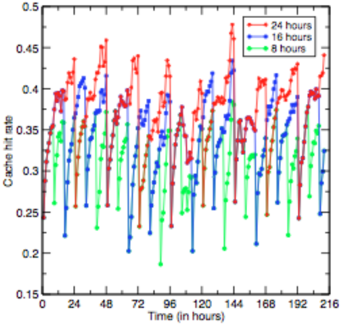
Flushing

- Flush the content of the cache and re-warm it from scratch



But...

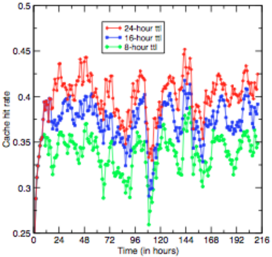
- ...it can lead to significant degradation of hit rates due to many compulsory misses



- High query traffic to the back-end search clusters
- Deep impact on result quality

Time-to-live

- Amount of time the search engine is allowed to serve a given entry from the cache
- An entry is said to be *expired* if $t_{current} - t_{last\_update} > t_{ttl}$
- Every hit on an expired entry is treated as a **miss**
- This approach sets an upper-bound on the age of a hit
- Does not prevent the search engine from serving stale results!



- Not optimal, but easy to implement
- **Problem**: Negative impact of expired entries on hit rate

Refreshing

- **Idea**: use idle cycles of the back-end query processors to refresh expired cache entries
- Two major benefits:
  1. Increase hit rates by reducing the number of misses due to requests on expired entries
  2. The number of user queries hitting the back-end search clusters drops
- A refresh mechanism requires a policy to select entries to refresh and order them
- **MAX** Hit rate + **MIN** Average age
- Several possible criteria for selecting: frequency of the query, recency of the query, cost of processing the query at the back-end, and the probability of a change in the cached results.

Selection queries to refresh

- They chose to give higher priority to "hotter" and "older" queries
- Two-dimensional bucket array, to keep track of **temperature** and **age**
- T Number of temperatures buckets, A Number of age buckets
- The hottest temperature and the freshest age are both zero.
  1. they initially add a query to bucket $(T-1, 0)$
  2. they increase the age of cached entries by shifting the buckets along the age dimension as times elapses
- They determine the interval between age shifts using two input parameters:
  1. The number of age buckets A
  2. Number of **singleton requests**

- They adjust the temperature according to the frequency of occurrence
- Lazy update $\rightarrow$ they recompute the temperature of a query upon either a hit or a refresh attempt
- How to select queries to refresh? they use a policy that selects hotter and older queries first
    1. For every temperature τ and age α, compute the value of $s = (T - \tau) \times \alpha$
    2. Order buckets according to decreasing order of the value of s

How many queries to refresh at a time?

Refresh-rate adjustment

- **Latency** is the main guidance in cache refresh rate adjustment
- Average latency (**tick latency**) is generated after every **tick**
- For every cache node, they have an expected latency range, $[L \cdots H]$
- Target number of queries, $T =$ User queries + Refresh queries

Algorithm 1 Adjust target number of queries ($\mathcal{T}$)

```
lat ← getTickLatency()
Q ← getNumQueriesProcessed()
if lat < L and Q ≥ T then
   T ← min(T + δ1, M)
else if lat > H and Q ≤ T then
   T ← max(T − δ2, 0)
else
   T ← T
end if
```

Simulation parameters

- Infinite cache
- 8, 16, and 24 hours for the TTL values
- Number of refreshes: PST[1] - Incoming query rate
- MRA minimum refresh age. Depends on the TTL parameter and equals to TTL/2, TTL/4, or TTL/8.

[1] peak sustainable throughput

Baseline algorithms

- Simple refresh algorithm: which seeks in the LRU queue the entries stored longer than MRA.
- Three issues:
  1. LRU queue grows: some entries are never refreshed
  2. Traversing the entire LRU queue is too costly in practice
  3. Most entries right after the head are unnecessarily scanned many times since they are either fresh or have been just scanned
- A slight change: **cyclic refresh**, a scan continues from where the previous scan terminated
- Baseline:
  1. cyclic refresh algorithm (**cyclic refresh**)
  2. TTL-based algorithm with no refreshes (**no refresh**)

Comparison

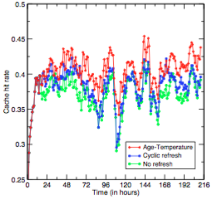


Figure 11: Hit rate with different policies.

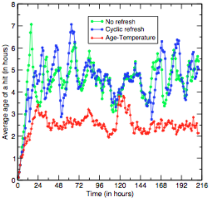


Figure 12: Hit age with different policies.

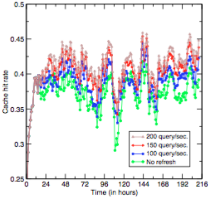


Figure 13: Hit rate with varying peak sustainable throughput.

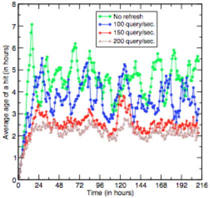


Figure 14: Hit age with varying peak sustainable throughput.

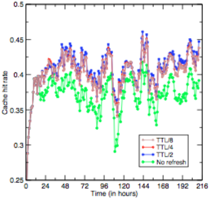


Figure 15: Hit rate as minimum refresh age varies.

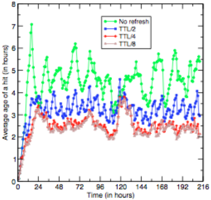


Figure 16: Hit age as minimum refresh age varies.

Table 1: Hit rates averaged over the entire query log

				Cyclic Refresh			Age-Temperature		
TTL	MRA	No Refresh	Flushing	PST=100	PST=150	PST=200	PST=100	PST=150	PST=200
8	TTL/2	0.337	0.311	0.338	0.343	0.35	0.352	0.373	0.388
	TTL/4	0.337	0.311	0.338	0.343	0.349	0.352	0.372	0.381
	TTL/8	0.337	0.311	0.338	0.343	0.348	0.352	0.372	0.381
16	TTL/2	0.372	0.345	0.374	0.382	0.395	0.389	0.407	0.41
	TTL/4	0.372	0.345	0.374	0.382	0.392	0.389	0.403	0.41
	TTL/8	0.372	0.345	0.374	0.381	0.39	0.389	0.402	0.407
24	TTL/2	0.398	0.369	0.401	0.409	0.424	0.417	0.426	0.427
	TTL/4	0.398	0.369	0.401	0.409	0.42	0.416	0.426	0.428
	TTL/8	0.398	0.369	0.401	0.41	0.422	0.416	0.424	0.427

Table 2: Hit ages averaged over the entire query log

				Cyclic Refresh			Age-Temperature		
TTL	MRA	No Refresh	Flushing	PST=100	PST=150	PST=200	PST=100	PST=150	PST=200
8	TTL/2	2.079	1.36	2.086	2.11	2.141	1.954	1.595	1.484
	TTL/4	2.079	1.36	2.077	2.097	2.13	1.953	1.448	1.122
	TTL/8	2.079	1.36	2.079	2.112	2.129	1.953	1.447	1.089
16	TTL/2	4.488	3.121	4.513	4.592	4.686	3.773	3.209	3.11
	TTL/4	4.488	3.121	4.504	4.511	4.611	3.632	2.508	2.147
	TTL/8	4.488	3.121	4.484	4.492	4.593	3.627	2.389	1.915
24	TTL/2	7.51	5.332	7.557	7.5	7.225	5.791	5.092	5.047
	TTL/4	7.51	5.332	7.556	7.415	7.286	5.329	3.826	3.524
	TTL/8	7.51	5.332	7.517	7.454	7.317	5.322	3.525	3.009

Production experience

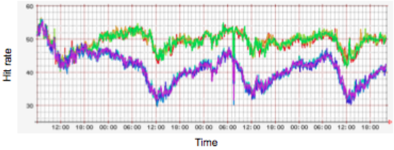


Figure 17: Hit rate in production (%).

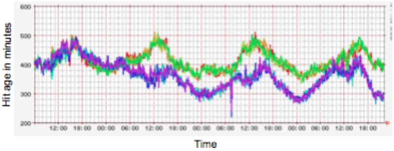


Figure 18: Hit age in production.

- 3 days
- The absolute difference is higher than 10% at several points
- The average hit rate with and without refreshes is 49.2% and 41.0%
- The TTL they use for these nodes is 18 hours and the average hit age with and without refreshes is 411.8 and 362.3 minutes

Observations

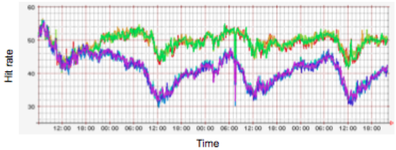


Figure 17: Hit rate in production (%).

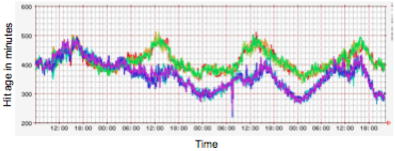


Figure 18: Hit age in production.

1. Hit rate difference between refreshing and not refreshing is between 7% and 10% in production and at most 5% in our simulations
2. Hit ages are higher when refreshing in production due to more conservative refreshing (it is possible to reduce the average hit age by refreshing more aggressively)

Degradation

- Back-end nodes may experience workload spikes due to various unpredictable reasons
- Based on the degree of degradation, they can find suitable TTLs for cache entries such that results with a lower degree of degradation receive a longer TTL
- Results with a high degree of degradation are given a higher refresh priority

Conclusions

- **New problem**: keeping cached results consistent with the search engines index while sustaining a high hit rate
- Flushing the cache is not efficient and they propose a **TTL-based strategy** to expire cache entries
- TTL parameter improves the average hit age with a loss in hit rate
- To improve hit rate and freshness, they introduce a refresh mechanism based on **access frequency** and **age of cached entries**

Thank you