Technical Report of cache

Peisheng Wang

December 9, 2008

Abstract

Cache is a vital component of a search engine system. Most cache system resides in memory and use list for storage of the information of cache item for replacement algorithm. However, this Cache, named MFCache resides in both memory and file for scalability and use RB-tree to store the caching information. It is also a policy-based in the form of C++ template class. With RB-tree structure, all kinds of replacement algorithm can be easily implemented under replacement policy. With storage policy, combination of hashes (memory-based hash and file-based hash) can be used for storage. With Capacity policy, large size of Cache can be applied. With Thread-safe policy, multi-thread or single thread application using this MFCache is available.

MFCache is equivalent to MCache when it only resides in memory. And we also add basic fastest memory cache MLRUCache, which using list as CacheInfo container. On constrast to MLRUCache, MCache has low efficiency and uses more memory, but it has higher hit ratio with the same cache size.

We also implement CacheDB, which is based linear hashtable or extendbile hashtable with MCache/MLRCache caching most of the items being used in memory such that it provides fast lookup to most retrieval calls.

Testing result shows that MFCache and MCache are highly effective.

Contents

1 Document History

Date	Author	Description
2008-12-05	Peisheng Wang	Revision to follow the TR format

2 Introduction

Nowadays, a search engines have to answer thousands of queries per second with interactive response time. Usually a single query may require the processing of hundreds of megabytes or more of index data. To keep up with this immense workload, hundreds or thousands of machines are employed, and a number of techniques such as caching, index compression, and index and query pruning are used to improve scalability.

In a realistic search engine, query results, inverted list, intersection of inverted list, UID related info, Content of web page, even connection (like TCP) and some computation can be cached to reduce responding time and improve the efficiency dramatically.

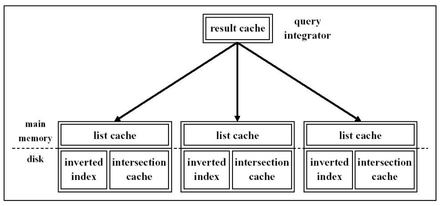

Figure 1: Three-level caching architecture with result caching at the query integrator, list caching in the memory of each node, and intersection caching on disk

In [**?**], a three-level caching architecture of Search Engine is investigated, which including result cache in query integrator(query level), list cache (term level) at each node, and intersection of corresponding inverted list (for multiple words query) . And it has improved the query throughput a lot. In addition, a two-level caching architecture is introduced in [**?**]

Therefore, it is admirable to design a reusable and scalable Cache that can deal with these caching above in a Search Engine.

In section 2, schedule of this project is presented.

In section 3, caching mechanism is introduced, including operation on Cache and container for caching information.

In section 4, requirement of a Cache is presented.

In section 5, an usable and scalable Cache design is presented and its experiment result is given in section 6. Further work is proposed in Section 7. And in section 8, Conclusion is presented.

3 Caching mechanism

Why does caching works? Reference locality [**?**]. Take query for example, many queries or terms would repeat more than once, even some would be very popular and repeat thousands of time. If the previous query result resides in Cache, then it will be very fast to get the result from cache for the next same query, i.e. a hit occurs. And hit ratio is the most important factor for a Cache. Given an infinite Cache, if the average repeated time is $n(n > 1)$, then the Hit Ratio is

$$1 - \frac{1}{n}$$

However, Cache Size can't be infinite. The Hit Ratio is determined by Temporal Locality Reference (TLR), i.e. the same queries repeatedly summited within small time interval (lifetime of a item in cache) .

when Cache is full, it needs to evict some item for new queries. And replacement algorithms(including admission policy and evicting policy) to find one item that most unwanted are introduced as follows:

3.1 Caching algorithm

- traditional

  1. LRU: to replace the least recently used item from the cache.

2. LFU: to replace the least frequently used item from the cache.
3. SLRU[1]: the item hits has higher priority than the item not hits, to replace the least recently used item not hits from the cache.
4. Pitkow/Recker[?] : to replace the document with largest size in the same day.

- Content base

  1. SIZE[?]: to replace the document with largest size.
  2. LRU-min[?]: to make the document needed to replaced least, using LRU for the documents with size at least S , S/2, and so on.
  3. LRU-Threshold[?]: like LRU, but the document size exceeds threshold can't be cached.
  4. Lowest Latency First[?]: to replace the document with lowest latency.

There are also price-based replacement algorithm using a price function to evaluate replacement price for each item in cache, such as Hybrid [?], Lowest Relative Value [?], Least Normalized Cost Replacement(LNCR) [?], Bolot [?], size-adjust LRU [?].

To maximize the Hit Ratio, many replacement algorithms have been studied. But no one algorithm is superior to any other algorithm under all circumstance. Different caching algorithm should be implemented according to the different visiting pattern to cache. Moreover, in [?], Predictive caching and prefectching technique are also introduced.

3.2 Caching Consistence

If one item resides in cache for a long time, then its content may be out of date. So it need to check periodically whether an item is overdue.

Above all, main method of an CacheManager are as follows (Note that, we use hash for storage, i.e. every item is presented by a key-value pair.):

3.2.1 Operation on CacheManager

On cache items (for our storage policy, it is the operation on Cache Hash):

- find(item)
- delete(item)
- insert(item)

On CacheInfo (CacheInfo stores the information of the corresponding cache item forreplacement algorithm.)

- find(min): return the handle of the item that we want to evict from CacheHash.
- update(key): update the information of the corresponding cache item for caching algorithm, which depends on the container of CacheInfo.

Table 1: Comparison of Containers for CacheInfo

Container	find(min)	update(key)
list	O(1)	O(1)(for LRU) O(n)(for LFU)
RB tree	O(1)	O(logn)
Priority-queue	O(1)	O(n)

3.2.2 Container for CacheInfo

From table 1, for list, it is Only good for LRU, but not easy to implement other replacement algorithm. For priority-queue, it is time consuming for update(key). For RB tree, it suits to all kinds of replacement algorithm. Therefore we choose RB tree as CacheInfo container.

3.3 Cache Storage

As is known that , Hit Ratio increases with cache size. To achieve the maximum capacity ofcache, a hybrid hash that consists of a memory-base hash and a file-base hash is highly recommended. There are 4 hash candidates, M_EH, M_LH, F_LH,F_EH.

4 MFCache Requirement

4.1 Key-Value types

MFCache deals with two types of data: Key and Value, where Key is a handle for Value. Given a Key, a Value must be returned if such cache item exists. Otherwise, returns false. Since In a real search engine, many things can be cached, the Cache should can deal with all kinds of Key-Value pair.

4.2 Cache policies

The MFCache Should have several configurable policies, including Replacement Policy, Storage Policy , Capacity Policy , Thread Policy. Note that, MLRU-Cache doesn't have Replacement Policy.

4.3 Replacement Policy

Replacement Policy determines what replacement policy to use. Among others, LRU and LFU must be provided for a choice. When the cache can not store any more cache items (key-value pairs), an item must be purged out of the cache storage in order to give a room for a new item which will be more likely to be accessed. SLRU is combination of LRU and LFU.

4.4 Storage Policy

Storage Policy determines the proportion of memory and file in terms of cache items in the cache storage. MFCache implements the memory cache storage using a memory-based hash table and file cache storage using a file- based hash table. A typical Storage Policy will store most cache items that are to be accessed more frequently in memory while putting the rest in file. The exact proportion will depend on the entire cache capacity and the availability of memory.

4.5 Capacity Policy

Capacity Policy determines the cache capacity, the total number of cache items Cache Manager can keep. A constant value is a legitimate Capacity Policy, fixing the maximum number of cache items in the system. However, the cache capacity can vary dynamically depending on the availability of memory and the data size that the uppe r application layer deals with.

4.6 Thread Policy

Thread Policy determines the thread-safety level of the MFCache system. No Thread Policy, for example, implements no thread policy but provides higher performance. If multiple clients access to MFCache at the same time, for example, inserting/removing items while searching for the same items, concurrency control is a key issue. Thread Policy implements a proper concurrency control depending on its policy.

5 Cache design

The MFCache system consists of these classes below:

```
class CacheHash;
class CacheExtHash;//derived from CacheHash.
struct CacheInfo;
struct xxxCmp; //for different replacement policy.
class CacheContainer; //storage of cacheInfo

class MFCache; //it resides both in memory and file.
class MCache;  //it only resides only in memory.
class MLRUCache;  //basic cache resides only in memory
   using.
class CacheDB; //File hash with a MCache as storage.
```

5.1 CacheHash

CacheManager implements its storage using hash tables. We implement a hybrid hash that combines both M_EH (memory-based extensible hash) and F_EH(file-based extensible hash).

We declare CacheExtHash as

```
template <class KeyType, class DataType, class FirstHash,
     class SecondHash>
class CacheExtHash:public CacheHash<KeyType, DataType>
```

Where FirstHash can be M_EH or M_LH, SecondHash can be F_EH, or F_LH. It has those private members as follows:

```
FirstHash memHash_;
SecondHash fileHash_;
float ratio_;
unsigned int  hashSize_;
```

To find an item in the hash, we use

```
find( key)
  if not find in memory Hash
     find in file Hash
```

To delete an item in the hash, we use

```
del( key)
  if  not delete from memory Hash
     delete from file Hash
```

To insert an item to the hash, we use

```
insert(data)
 if  find(key) return 1;
     else if memory hash  not full
         insert into memory hash
     else  if file hash not full
         insert into file hash
     else
         hash is full
```

We also provide dump method.

```
dump(key)
  if key in memory hash and file hash not full
     dump key from memory to file
  if key in file hash and memory hash is not full
     dump key from file to memory
```

5.2 CacheInfo

We use CacheInfo to store the necessary info for Caching Algorithm.

```
struct CacheInfo
```

CacheInfo provides necessary information for Replacement Algorithm. Any item in CacheHash has only one corresponding CacheInfo, and vice verse. It contains these members below:

```
KeyType key;
size_t  docSize;
bool    isHit;
time_t LastAccessTime;
time_t FirstAccessTime;
time_t TimeToLive;
unsigned int iCount;
```

Through these info in CacheInfo above, we can implement all kinds of Replacement algorithm, including LRU, LFU, SLRU and so on. To implement different algorithms, we only need to provide different comparisons object between CacheInfo objects.

For LRU, the comparison is

```
bool operator() ( CacheInfo &  lhs,   CacheInfo & rhs)
{
return (lhs.LastAccessTime < rhs.LastAccessTime)
        || (lhs.LastAccessTime == rhs.LastAccessTime)
            && (lhs.key < rhs.key) );
}

bool operator() ( CacheInfo &  lhs,   CacheInfo & rhs)
{
return (lhs.LastAccessTime < rhs.LastAccessTime)
        ||(lhs.LastAccessTime = = rhs.LastAccessTime) && (
            lhs.iCount == rhs.iCount) )
        ||( (lhs.iCount == rhs.iCount)
          && ( lhs.LastAccessTime = rhs.LastAccessTime )
          && (lhs.key < rhs.key) );
}
```

For LFU, the comparison is

```
bool operator() ( CacheInfo &  lhs,   CacheInfo & rhs)
{
return (lhs.iCount < rhs.iCount)
        ||( (lhs.iCount == rhs.iCount)
           && ( lhs.LastAccessTime <rhs.LastAccessTime ) )
        ||( (lhs.iCount == rhs.iCount)
           && ( lhs.LastAccessTime = rhs.LastAccessTime )
           && (lhs.key < rhs.key) );
}
```

Hence other Replacement algorithms can also be implemented this way.

5.3 CacheContainer

We store the CacheInfo objects in map(of STL) for implementing caching algorithm. The reason why we choose data structure map is that, map is implemented by Red-Black tree, and keys are sorted.

```
typedef map<CacheInfo<KeyType>, bool, ReplacementPolicy >
    CacheInfoKeyMap
CacheInfoKeyMap     keyInfoMap_;

typedef hash_map<KeyType, CacheInfo<KeyType>,
   ystring_hash> CmHashMap;
CmHashMap CacheInfoHash_;
```

We map also CacheInfo to bool variable isRefresh, where isRefresh can be used to determine whether a CacheInfo need to be purged out, which can be used for consistence mechanism.

CacheInfo -¿ isRefresh?

We also map key to CacheInfo for updating the information of key-value pairs for caching algorithm.

5.4 MFCache

The main class is Class MFCache

We declare class CachaManager as a template class.

```
template <class KeyType, class DataType, class
    ReplacementPolicy, class FirstHash, Class
SecondHash, class ThreadSafeLock=ReadWriteLock> class
    MFCache{}
```

So the CacheManager can deal with different type of key-data pair using different Replacement Policies and using different storage policies through template template parameters. It has private members

```
CacheExtHash<KeyType, DataType, FirstHash, SecondHash>
    hash_;
float ratio_;
unsigned int cacheSize_;
ThreadSafeLock lock_;
CacheContainer cacheContainer_;
```

5.5 interface of Caches

To find an item from the cache

```
bool getValue(key)
 if hash_.find (key)
    update the corresponding CacheInfo
    return TRUE
 else
    return FALSE
```

To insert an item into the cache.

```
insert(value)
 if not hash_.insert(key)
    delete  the items that the head of KeyInfoMap
        indicates
    delete the corresponding CacheInfo;
 if hash_.insert(key);
    insert the corresponding CacheInfo into CacheInfoMap
```

One of the most important methods is

```
getValueWithInsert(key, value)
 if getValue(key)
 else
    insert(value)
```

and MCache is MFCache with $ratio = 1.0$.

5.6 MLRUCache

For MLRUCache, we use std::list to store all the keys. Everytime, we insert an item into the cache, if it hits, we just delete it from the list, and insert it into the end of the list, and if not hits, we just insert it into the end of of the list. To save memory usage and achieve fast access of list items, we wrap DataType with iterator of the list.

```
typedef list<KeyType> CacheInfoList;
typedef typename list<KeyType> :: iterator LIT;

template<class K, class D>
struct _CachedData {
   D data;
   LIT lit;
   const K& get_key() const{
       return (const K&)data.get_key();
   }
};
typedef _CachedData<KeyType, DataType> CachedData;
typedef ExtendibleHash<KeyType, CachedData, NullLock>
    extHash;
typedef LinearHashTable<KeyType, CachedData, NullLock>
    linHash;
```

5.7 CacheDB

It is persistent in that it stores all the key-value pairs in file(using linear file hash). Plus, it also caches most of the items being used in memory such that it provides fast lookup to most of the retrieval calls. It has all the replacement policy, storage policy, and others. CacheDB can be used as our base DB for key-value pairs. There are many open source platforms like this, such as berkeley DB and gdbm. Our version is based on Linear Hashtable and supports efficient caching explicitly (instead of relying on OS virtual memory/swapping system). It also support multi-threads and locking/concurrency.

6 MFCache experiment

We read words sequentially from data set Modlewis.dat that consist of thousands of articles , and append the word with random number within 0 100 , then stores the words in wordlist. The total number of words is 146287 words. The number of unique words are 363684. We take the words as the input key, then call

```
//hit, return 1; not hit, return 0
bool getValueWithInsert(key, value)
```

6.1 F_EH vs F_LH testing

It sets CacheSize=10000, ratio = 0.4, replace = LRU, and no dump. see table 2.

Table 2: F_EH vs F_LH testing

hash	FEH	FLH
elapsed	13	6
hit ratio	1395475/1462877	1390335/1462877
elapsed	16	7
hit ratio	1396031/1462877	1390393/1462877
elapsed	17	7
hit ratio	1396059/1462877	1390371/1462877

Table 3: Dump Option testing

dump option	elapsed	hit ratio
0	21	677658/1462877
1	23	679437/1462877
2	21	678727/1462877

6.2 Dump Option testing

It sets CacheSize = 20000, ratio=0.4 ,replace = LRU, FirstHash = ExtendibleHashMemory, SecondHash = LinearFileHash. See table 3.

6.3 Replacement algorithm testing

It sets CacheSize= 50000, ratio= 0.4, FirstHash = ExtendibleHashMemory, SecondHash = LinearFileHash.See table 4.

6.4 CacheSize testing

It sets ratio= 0.4, replace = slru, FirstHash = ExtendibleHashMemory, SecondHash = LinearFileHash. See table 5.

6.5 Ratio testing

It sets CacheSize= 50000, replace = slru, FirstHash = ExtendibleHashMemory, SecondHash = LinearFileHash. See table 6.

6.6 Multi-thread testing

Divide wordlist into 15 files, and set cacheSize = 20000, ratio = 0.4, replace=lru, 15 threads deal with one file respectively. See table 7.

Table 4: Replacement algorithm testing

replacement	elapsed	hit ratio
LRU	17	837459/1462877
LFU	17	878781/1462877
SLRU	15	878096/1462877

Table 5: CacheSize testing

cachesize	elapsed	hit ratio
5000	15	441761/1462877
10000	16	589157/1462877
20000	17	722027/1462877
50000	16	879200/1462877
100000	16	986347/1462877
200000	16	1066954/1462877

Table 6: Ratio testing

ratio	elapsed	hit ratio
0	20	880425/1462877
0.2	18	880083/1462877
0.4	16	878725/1462877
0.5	16	878851/1462877
1.0	11	876010/1462877

6.7 Experiment analysis

- LearFileHash outperforms ExtendibleHashFile.
- In our testing, efficiency is not improved with dump option. It is better to conduct dump when CPU is idle.
- Performance algorithm: SLRU ¿LFU ¿LRU
- The larger Cache Size , the higher Hit Ratio, but less marginally increase on Hit ratio. While efficiency don't decline
- The higher ratio(memory vs file), the less efficiency, only very little decline in hit ratio.

7 Further enhancement

Distributed Cache.

8 Conclusion

Cache is a very important component in search engine. In a realistic search engine, query results, inverted list, intersection of inverted list, UID related info, even connection (like TCP) and some computation can be cached to reduce responding time and improve the efficiency. Our objective is to design a

Table 7: multi-thread testing

elapsed	hit ratio
20	738492/1462854

MFCache that can deal with these caching above, and also take the issues like caching polices, caching consistence into account.

MFCache we designs resides in both memory and file for scalability and use RB-tree to store the cache information. It is also a policy-based CacheManager in the form of C++ template class. With RB-tree structure, all kinds of replacement algorithm can be easily implemented under replacement policy. With storage policy, combination of hashes(memory-based hash and file-based hash) can be used for storage. With Capacity policy, large size of Cache can be applied. With Thread-safe policy, multi-thread or single thread application using this MFCache is available.

Testing result shows that these Caches has high efficiency.

A Schedule

A.1 Stage1

Totol duration : 10 weeks

Milestone

1. Preparation
   - Duration: 1 week
   - People in charge: Peisheng Wang
   - Description
     Learn hash, linear Hash, extensible hash. Read relevant paper and finish a survey about Cache Manger in search engine.
2. initial implementation
   - Duration: 2 weeks
   - People in charge: Peisheng Wang
   - Description
     Learn multiple-thread programming, policy-based design, template programming. Finish a prototype.
3. Testing, improvement, and documentation
   - Duration 4 weeks
   - people in charge: Peisheng Wang
   - Description
     - Finish design documentation of Cache Manager.
     - Implement thread-safe policy.
     - Multi thread testing. Thoroughly testing on all kind of policies.
     - Adjust code style. Make it doxygen ready.
     - Adjust to meet the new requirement. No Cache manager, but MCache, MFCache and CacheDB.
     - Refactor. Add class **CacheContainer**. Integrate them into SF1LIB library.

    - Technical report

4. Maintenance
    - Duration 2 weeks
    - People in charge: Peisheng Wang
    - Description
        - Add getEfficency method. Add get memory usage info method. To monitor the performance of Cache.
        - Add serialization method for M_LH and M_EH.
        - Add serialization method for MCache and MFCache. When MCache or MFCache may fail at runtime, it can be recovered when running it again
        - Add **CopyTo(AccessMethods& am)** for M_LH and M_EH ,for M_LH and M_EH is not copyable with thread safe policy.

5. Maintenance1
    - Duration: 1 weeks
    - People in charge: Peisheng Wang
    - Description Add MLRUCache.

A.2 Stage2

MileStone	Finish Date	In Charge	Description
Maintenance	2008-12-05	Peisheng Wang	Reorganize the structure of Cache, add thorough multithread-testing code, adjust the TR

B How to use Cache,MFCache and CacheDB?

B.1 example

For usage of these codes, please see example in src directory:

- t_MCache.cc, example for MCache.
- t_MLRUCache.cc, example for MLRUCache.
- t_MFCache.cc, example for MFCache.
- t_CacheDB.cc, example for CacheDB.
- t_mfserialize.cc and t_mserialize.cc, example for MCache and MFCache serialization.
- t_mul_cache.cc, example for multi-thread testing of caches and CacheDB.

Index