NTNU
Det skapende universitet

Håndtering av page cache i Derby

Olav Engelsåstrø
Øyvind Reinsberg

Master i datateknikk
Oppgaven levert: Juni 2006
Hovedveileder: Svein Erik Bratsberg, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Oppgaven går ut på å studere Derbys mekanisme for buffer og page caching, og å se på altervative løsninger.
Oppgaven vil bestå av en studie av Derbys nåverende cachesystem, litteraturstudie av alternative cacheteknikker og evt. implementasjon av en eller flere teknikker.

Oppgaven gitt: 20. januar 2006
Hovedveileder: Svein Erik Bratsberg, IDI

Contents

List of Figures

Abstract

The open source RDBMS Derby (version 10.1) is without a dedicated page cache algorithm. Instead it uses the general purpose algorithm Clock for all types of caching. Taking the access patterns of page caching into consideration when designing the page cache algorithm will give performance benefits. This thesis begins with a study of page caching algorithms. Two closly related algorithms were then chosen to be implemented. We chose LRU and LRU/2Q. The former serves as a basic plattform for creating a stable version, helping to figure out the interfaces and constraints given by Derby. With LRU up and running we implemented the more complex LRU/2Q. All three algorithms (Clock, LRU and LRU/2Q) were then tested with a varity of parameters including page sizes, cache size and number of records per page. The results show that both LRU and LRU/2Q outperforming Clock on all test cases.

Chapter 1

Introduction

Apache Derby is an open source database system written in pure Java, distributed under the Apache license, version 2.0. It features full database functionality while keeping a small footprint. It can run embedded in applications or as a network server. Currently Derby is using the clock page replacement algorithm for swapping pages at runtime. The background for this study is the thesis that the clock is not a good algorithm for a relational database system.

Chapter 2 gives an overview of the Derby database system. Chapter 3 gives a theoretical introduction to the page replacement problem, and discusses different existing algorithms. Chapter 4 describes the implementation issues for the implemented algorithms, LRU and LRU-2Q. It also describes these algorithms in detail. In chapter 5 we verify that our implementation of LRU-2Q is correct. In chapter 6 we do several tests using different parameters, to measure the performance of both clock, LRU and LRU-2Q, and analyze the results.

Chapter 2

Derby Overview

2.1 Introduction to Derby

2.1.1 About Derby

Derby is an open-source RDBM system. It supports two modes. One with the standard client-server architecture, but the most interesting is the embedded mode. You can get a full database system within a Java solution by just adding a jar file of 2MB to the classpath of your project. You connect to the database, as you would with any other Java system, with a JDBC driver. This is very interesting for people who are developing for small units like mobile devices, PDAs, etc. The system will run on every hardware and device that has a Java virtual machine.

2.1.2 History

The Derby database was originally named JDBMS, developed by Cloudscape, Inc, and released in 1997. The product was then renamed Cloudscape. Informix Software, Inc., acquired Cloudscape, Inc in 1999. In 2001 IBM acquired the database department of Informix Software, including Cloudscape and renamed to IBM Cloudscape. IBM focused the development on embedded use. IBM contributed the code to Apache Software Foundation in August 2004 as Derby. In July 2005 Derby graduated from being an incubator project to a complete Apache DB project. The database system consists currently of about 500 000 lines of code, and 1500 classes and interfaces. IBM still sells Cloudscape, but it is only Derby with support. In December 2005, Sun decided to ship Derby as open Java DB as a part of their Sun Java Enterprise System.

2.1.3 Standards

Derby supports the SQL standards SQL92, SQL99, SQL2003 and SQL/XML. The SQL dialect in use, is closely related to IBM's DB2. Derby can be run on Java J2SE

1.3 and 1.4, J2EE 1.3 and 1.4 and J2ME. When run in Network Server mode, users connect using a DRDA connection.

DRDA (Distributed Relational Database Architecture) was developed by the Database Interoperability Working Group for Open Group[4]. DRDA is a standardized database protocol, currently supported by all IBM DB2 databases and Cloudscape/Derby. The group was supported by the likes of Fujitsu, IBM, Microsoft and SCO, but the group is now closed and no further work will be done on the protocol by Open Group.

2.1.4 Constraints

The size of a Derby database is limited to a single logical disk, only the log may be placed on a different volume. It is, however, possible to use RAID to simulate larger logical disks.

Derby runs fine with 20-30 active connections and 100-500 updates per second. If you have many connections and transactions you might get into memory trouble. If you start a Java application, the default maximum memory allowed is usually set to 64MB. This can be increased by using the -Xmx**size** option when running the application.

Since Derby runs inside the JVM, Derby can easily take advantage of using multiple CPUs. Java has support for threading and synchronization. How well it scales with the number of CPUs is unknown, but it should scale as well as Java scale.

2.1.5 Activity

Derby is currently being actively developed. From September 2005 to June 2006 almost 15000 emails were sent to the Derby mailing lists. Both Sun and IBM have full-time developers devoted to Derby. There are about 20 developers working with derby, and about 10 committers[1].

2.2 Architecture Overview

The Derby homepage[2] presents two alternative views for the Derby architecture called Monitor view and Layer/Box view Figure 2.1. The Layer/Box view is the most comprehensive and descriptive for Derby's architecture. Further references to the Derby architecture will be based on this view. The Layer/Box view divides Derby into four main code areas.

JDBC

The JDBC layer is the entry point for connections and queries into Derby. The JDBC consists of implementations of JDBC interfaces and support classes. For doc-

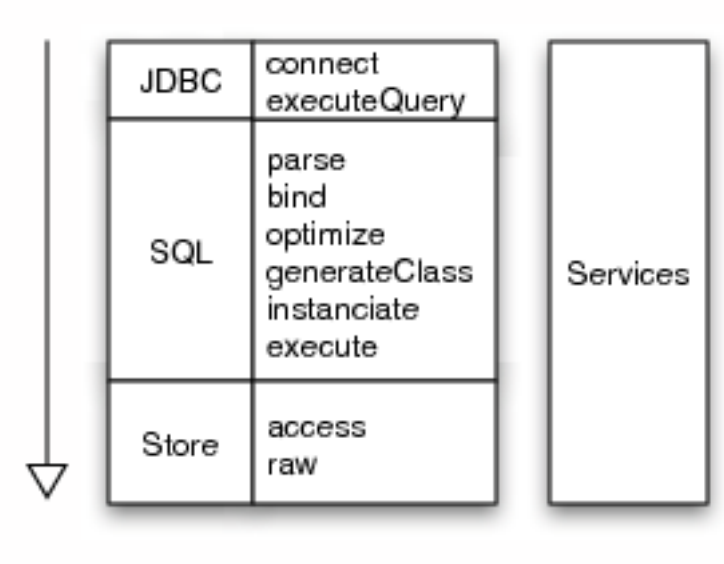

Figure 2.1: Derby architecture layer view

umentation on JDBC please see [3]

SQL

The SQL layer is divided into two sub-layers, compilation and execution. SQL Compilation consists of:

- Parsing of the query into a node tree.
- Binding node tree table objects to data dictionary.
- Optimizing the node tree, with respect to the access path.
- Generating a java class representing the statement plan.
- Loading and instantiating the generated class.

Execution consists of calling execute on the instance of the generated java class, returning a resultset. The resultset is responsible for interacting with the store, and filling the resultset when needed.

Store

The store layer is divided into two layers, access and raw.

- The access layer is the interface for the SQL layer. The access layer is responsible for scans, lookups, indexing, locking, transactions and more.

- The raw store is responsible for the pages and files where the database rows are stored.

Services

Services are program modules responsible for tasks like caching, locking and logging. The services are shared for all connections made to the database. Hence there is only one cache instance even though several clients may connect to the database at different times.

There are eight different caches in Derby. These are:

- PageCache
- ContainerCache
- ConglomerateDirectoryCache
- VMTypeIdCache
- TableDescriptorOIDCache
- TableDescriptorNameCache
- SPSNameDescriptorCache
- StatementCache

All these caches are created as different instances, but they all use the same generic caching algorithm, which is Clock, as of latest official release at present time, 10.1.2.1. The scope of this paper is only to look at the page caching function, so the rest is concentrated on the PageCache only.

Chapter 3

Page replacement algorithms

3.1 Cache Algorithms

Cache algorithms is a mature field of computer science. Its applications are many and cache algorithms are utilized in many disciplines of software engineering. Basic well known cache algorithms can be applied to most breeds of cache situations. But higher performance is achieved if the algorithm takes domain related specifics into account. Operating systems is one of the main software disciplines that rely on caching at multiple levels, and a good portion of the research into the area is affected by this. Databases uses cache algorithms for many purposes, including page replacement.

3.1.1 The page replacement problem

Since the beginning of computers there has always been two types of storage. Data which is in use by the CPU is placed in the main memory(RAM). This storage is quite expensive, fast and is removed when the computer is shut down. Therefore computers have a secondary storage, which is usually a hard drive. Hard drives are cheaper per storage unit, far slower than RAM but the content is kept even though the computer crashes or shut down.

Because it is slow to copy data out of hard drives we want to keep content in the main memory as long as they are used, and throw it out when it is not. The policy of choosing which page to be removed from memory when a new data item is needed and the memory is full, is called the page replacement algorithm.

In a DBMS the data units used are called "pages" and refers to the logical unit on the hard drive. Each page can contain one or more tuples, depending on the size of the tuples. Typical values for a page are from 4Kb to 64Kb. The memory space available to a DBMS is called the "cache". Typical databases are bigger than it is possible to fit in a database cache. Even though memory has become both cheaper and bigger, the data volumes has also increased.

3.1.2 The theoretical optimal page replacement algorithm

The optimal page replacement is actually quite simple. When a page has to be replaced, the page that will not be used in the longest time will be thrown out. In example, if one page will be accessed in 4000 accesses, and another in 5000 accesses, the one that will be accessed in 5000 will be replaced.

The problem with this approach, is that it is not possible to implement in a real-time system. This is simply because nor we, the system or anyone will know what accesses will happen in the future. The only time this can be used in a real-time system, is that if the exact same pattern is accessed two times. Then it is possible to use the knowledge found in the first run to find the optimal page replacement scheme. This, however, will happen very rarely in a multiuser environment as the databases. Users will execute queries at their own will.

It could though be useful to compare a page replacement algorithm to the optimal one, to measure the performance of it. This is however must be done manually or by a simulator afterwards for a special pattern.

3.1.3 Relational database page replacement algorithms

This section will look at the internals in cache algorithms in DBMSs. Managing a cache is performing two tasks, marking pages for replacement and doing the actual replacement in example reading/writing pages.

Page replacement in databases has many things in common with page replacement in operating systems. There are however some significant differences in the way pages are being accessed. While operating system accesses are usually just single pages, it is much more common for database queries to access data through scanning an entire tables, that very often is much bigger than the space available in the main memory. These scans will actually flush the cache, and result in a very low hit rate using standard caching algorithms.

A database page replacement algorithm can also usually handle more overhead than an operating system. This is due to the fact that an DBMSs are very complex and already has a lot of overhead.

Some of the characteristics of access patterns for DBMS queries are:

- Indices. Most databases use indices for faster access of data. The index pages are accessed far more frequently than ordinary data pages.
- Scans. Scanning an entire table for values. This could be operations such as "Get all records of a table" or aggregate queries like "Get the person with highest income".
- Transactions. The database system might be aware of future patterns, that might help the algorithm to know which page to throw out of the cache
- Co-related references - This is a typical access pattern seen in databases. A page gets multiple references over a short period of time. This is described in

more detail in section 3.1.4.

Page replacement algorithms can be divided into different categories:

- Stochastic - This includes basic algorithms like FIFO, CLOCK, LRU [5] and others. This category also includes more advanced algorithms, usually modifications over the basic variants. The advanced algorithms include more data structures, logic and have been developed to suit more special environments, such as the database environment. Strategies like LRU/K[7], LRU/2Q[8], ARC[10], CAR and CART[9] are examples of advanced stochastic algorithms.

- Hint passing algorithms - This line of algorithms cooperates with the query optimizer. The optimizer will pass hints to the cache manager. The cache manager will use these hints customize the caching strategy. One example of a hint passing algorithm is QLSM [6], which is described in more detail in section 3.2.

3.1.4 Co-related references

Co-related references is a typical access patteren for databases. The pattern consist of many following references to a page in a short period of time. This can happen if one were to, for instance, scan a table. All records in the table would then be accessed in turn. When all records have been accessed no more references are made. Pages that have completed a set of corelated references wastes cache space.

3.2 Alternatives

This section discusses different implementations of algorithms used for page replacement. We only describe a few selected algorithms here.

3.2.1 Stochastic algorithms

Clock

The clock algorithm is very simple. It contains a circular list of all pages kept in cache, as shown in figure 3.1. All pages keep a bit which tells if the page has been accessed since the last round. When a page replacement will happen, the algorithm checks the page of where the pointer is pointing. If the bit is 1, the bit is set to 0 and the pointer moves to the next page. If the bit is 0, that page is replaced with the new page. In case a page hit is found, the bit will just be set to 1.

Derby is currently using a slightly modified version of clock. The advantages of the clock, is its simple implementation and low overhead. It was however developed as an operating system algorithm, and is probably not the best for an advanced database system.

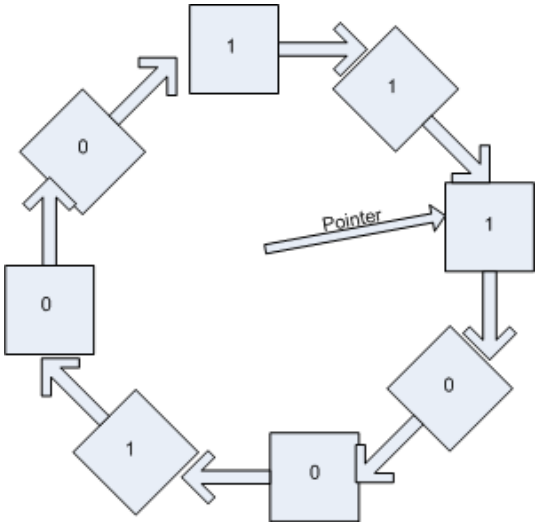

Figure 3.1: The clock replacement algorithm

CAR/ CART - Clock with Adaptive Replacement / Temporal

CAR is a variant of clock that requires two hits to promote to main buffer. CART also checks the time intervals between the hits, only allowing hits that are separated by a certain threshold. This filters for closely related cache hits, also known as co-related references. More about co-related references can be found in section 3.1.4.

Least Recently Used (LRU)

LRU is based on the idea that when a page has to be replaced, the page that has been unused for the longest time will be thrown out of the cache. The LRU is usually implemented using a linked list. The page that is being accessed is put in the front of the list, and if the page is not in the cache, the last page in the list is being deleted. An example of the LRU algorithm is shown in Figure 3.2.

The LRU has shown to be a good replacement algorithm. The problem is that to search for an item in a linked list is in worst case O(N). This will also be a case in all page misses. This issue can be solved by maintaining an additional data structure that keeps an overview of which pages is in the list. A hashtable with O(1) lookup is smart to use in this case.

3.2.2 Advanced Stochastic algorithms

LRU/K

LRU/K [7] is a modified version of LRU. The idea is to keep charge of the K most recent accesses of a page. K=1 is just the normal LRU. The most usual implementation of LRU/K is with K=2, so that the algorithm remembers the last 2 accesses of a page. The reason for this is that a lot of pages are just accessed once, and then

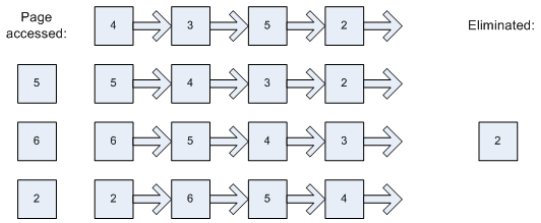

Figure 3.2: The LRU replacement algorithm

not accessed for a long time. This is the case of a table scan in a database system. The LRU/K maintains two lists. The items are placed in one the first time they are accessed. If they are accessed again while they are in this list, they are moved to another list, that is only for items accessed twice or more.

LRU/2Q

LRU/2Q is just a slightly modification of LRU/2. As presented in article [8], the basic LRU/2 algorithm is exhausting for the processor, since each page lookup is at log(N) work. They present a new algorithm with constant lookup time. Like CART, LRU/2Q also tests for co-related references, not allowing them to the main buffer. We will discuss LRU/2Q in detail in section 4.7.

LRU/2Q was implemented in Postgres 8.0.2 The Postgres community decided later (as of 8.1) to use a different algorithm, called the "Clock-Sweep"[11].

3.2.3 QLSM

Query Locality Set Model (QLSM)

QLSM was introduced in [6] as an alternative to manage the page cache. It is the only hint passing algorithm discussed here. It tries to predict the future page accesses by hints received from the parser when a query is being executed. QLSM forms its caching strategy to what kind of query is being made and how the relation is stored on disk. For instance a table scan will use a different strategy then an insert. Access is classified as one of the following: Straight sequential references, clustered sequential references, looping sequential references, independent random references, clustered random references, straight hierarchical and looping hierarchical references.

The QLSM is most likely a good approach for a DBMS page replacement strategy. Variations of this method is supported by bigger DBMS like Oracle and DB2. We will however not look in more detail to this strategy, since it would require a huge rewrite of Derby, and non of its competitors of lightweight databases(like Postgres and MySql) support this.

3.3 Choosing an Algorithm

The jungle of cache algorithms are huge, and it would be impossible for anyone to claim that one is the best for all kinds of use. It is always possible to tweak tests to be optimal, by adjusting parameters like cache size, database size, page size and using different kinds of statements. While some algorithm might be better in literature, it might not be the best when put into an existing database. We consider hint passing algorithms like QLSM to be over the top for this project. The open databases that we have looked into (MySql and PostgreSQL) use stochastic algorithms. Picking a stochastic algorithm to implement seem like a half chance choice between LRU/2Q and CART. These both share the idea to use two buffers. One for pages that are accessed once, and one for pages that are accessed two or more times over a longer period of time. It was not easy to know which one to select, since both are good solutions to our problem. But in the end we had to choose one to implement, so we went with LRU/2Q.

We will implemented two versions of LRU, basic LRU and LRU/2Q. LRU is well known, and LRU/2Q is used in other open databases like PostgreSQL 8.0.2. LRU is a simple, yet well performing algorithm. In addition to serving as its own test case, implementing LRU will serve as a testing ground for creating a page cache manager for Derby. Once a running version of LRU has been created the more complex LRU/2Q will be implemented. The implemented algorithms will be tested and compared to clock, the algorithm that is already implemented in Derby

Chapter 4

Design and Implementation

4.1 Introduction

Implementing the algorithm have two different and somewhat independent aspects. One is administration of the LRU structure. The second is compatibility with Derby code. The later will be the dominating task in creating a running version of the algorithm. Some of these aspects are discussed in 4.3. The motivation for implementing the basic LRU is mastering the Derby interfaces. When a running implementation has been created we can turn our attention to creating a more complex implementation. Adjustments to the LRU can then be made without affecting the compatibility. With this in mind we define our strategy.

1. Create a basic LRU implementation whose primary goal is compatibility with Derby. This implementation will serve as a base for further development.

2. Development of a higher complexity algorithm, LRU2/2Q.

4.2 Derby API interfaces and central classes

Interfaces in the Cache manager

The interfaces that must be implemented in a page cache manager for Derby are the following.

- CacheManager - This is a generic interface for all cache managers in Derby (like statement cache or page cache). The most important methods are find and release. Additional methods include resizing and remove.
  Path: org.apache.derby.iapi.services.cache.Cachemanager

- Serviceable - must be implemented by any class wanting to use Derbys internal daemon service. It has 3 methods performWork(),serviceASAP() and serviceImmediately(). A new cache manager is free to use these methods as needed.
  Path: org.apache.derby.iapi.services.deamon.Serviceable

Important classes and interfaces

These already implemented classes and interfaces are an important part of the cache service.

- PageKey - This class is used to identify a page on disk. It has two fields, containerId and pagenumber. It is the input key of page cachemangager methods like find and release.
  Path: org.apache.derby.iapi.store.raw.PageKey

- Cacheable - This is a generic interface for all items stored in any cache manager in Derby. Different cache types has different implementations of Cacheable. For a page cache manager the implementation will is StoredPage. The Cacheable have a field, "identity", that points to a cached object, in this case a page. The identity of a Cacheable may change when items in cache are replaced, but the Cacheable persists (except if downsizing the cache).
  Path: org.apache.derby.iapi.services.cache.Cacheable

- StoredPage - This class has methods for writing to disk and others required for page caching.
  Path: org.apache.derby.impl.store.raw.data.StoredPage

- SanityManager - The SanityManager class of Derby is the debug control center for the database engine. A build can be set to be sane or insane. The final variable DEBUG is set true if a build is sane. The class is static and so can be reached from anywhere. Its most common check is ASSERT, which checks a given condition. If the condition does not hold an exception is thrown. The SanityManager takes care of error reporting and handling.
  Path: org.apache.derby.iapi.services.sanity.SanityManager

- ClockFactory - Despite it's name ClockFactory works as a generic cache factory. It's name is probably due to the fact that at this point all cache managers in Derby are Clocks. One extra line of code in this class is needed to boot a new type of cache manager.
  Path:org.apache.derby.impl.services.cache.ClockFactory

4.3 Shared mechanics of LRU and LRU/2Q implementations in Derby

On of the challenges of creation a new cache algorithm for Derby lies in implementing interfaces and constraints given by Derby source code. We seek not to modify the interfaces. The interfaces are used for all kinds of caching. In Derby 10.1 only a single cache manager is used for all kinds of caching. The lesser impact the new page cache algorithm will have on these interfaces the better.

Some of the constraints these interfaces introduce are the following.

- keepCount - Items can not be removed from cache if they are referenced by Derby (other than the CacheManager). One will probably encounter situations where the most least recently used item is still referenced. This would have to be handled be traversing the list, trying to find another item. This both increases the complexity of the LRU and increases the chance of items being pinned in cache. In a worst case situation all items in the cache might be referenced. In this case the only option would be to grow the cache (we can not wait for pages to be released, this would break synchronization). The number of references to a Cacheable is referred to as the keepCount. The page cache implementation of Cacheable provides no mechanism for handling the keepCount. As a consequence of this the Cacheables will need some sort of holder items, or wrappers. Also the LRU logic have to be extended to deal with this. This is described in figure 4.2.

- Multithreading - The CacheManager interfaces declare which methods have to be thread safe. Thread safe segments of code blocks the object from other threads, and so imposes a time penalty. Synchronized code can not wait for other threads accessing the locked object, as waiting releases the locks.

LRUItem and keepCount

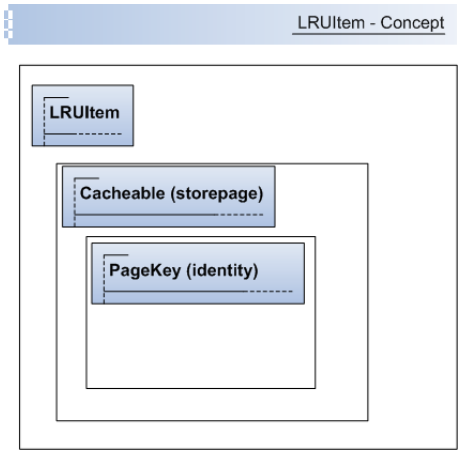

Figure 4.1: Hierarchy LRUItem and content

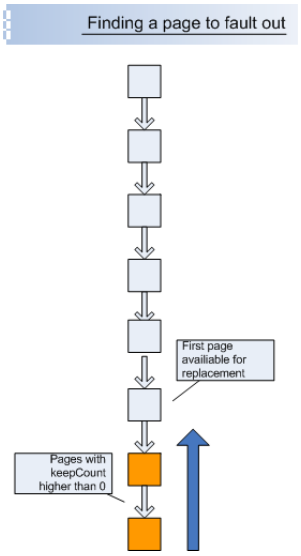

Figure 4.2: Finding a free item

The keepCount is the main reason why implementing LRU / LRU/2Q in derby, can not be implemented as straightforward LRU / LRU/2Q. LRUItems are requires to keep track of the keepCount of a cached object. The same implementation is used for both LRU and LRU/2Q. Figure 4.1 show the connection between LRUItems,

Cacheables and Identities. The LRUItem may be in one of three different states. LRUItems in state number 3 will be referred to as active items.

1. No Cacheable - This occurs only during initialization, and before the item is used for the for the first time.

2. Have Cacheable, without Identity - This state occurs if the LRUItem has been reclaimed for a new page or the page has been explicitly removed from cache. The Cacheable Objects are recycled for new Identities

3. Have Cacheable with Identity - This is an active cache item. When identity is set the Cacheable represents a page on disk.

The LRUItem holds fields that are required by the CacheManager implementation. The fields are

- keepCount - This represents the number of references that are held to the Cacheable from outside the cache manager. Holding a reference to a Cacheable blocks it from being evicted from the cache, and so references are removed as soon as possible. The keepCount is increased after the Cacheable is found with the Find method of CacheManager. It is decreased when released is called on a Cacheable.

- removeRequested - This boolean is a flag set when CacheManager.remove has been called on the Cacheable. The Cacheable can not be removed if its keepCount is over 0. Removing will have to wait for the keepCount to drop. This means releasing the synchronize lock. By setting the flag one prevents someone else from trying to remove the same Cacheable.

A page replacement operation in Derby LRU can not just pick the last page, but will have to traverse the list until it can find an item that has a keepCount of 0. This is handled in the same way for LRU, and LRU/2Qs lists aIn and Am. Figure 4.2 displays an example of this.

The findFreeItem method

This code segment shows the generic method used by both LRU and LRU/2Q to find the last item that can be evicted.

```
public LRUItem findFreeItem(LinkedList ll,int maxSize,HashMap
    hm, LinkedList overflow,HashMap OverflowHash,int
    overFlowSize) throws StandardException
{
  if(debug) System.out.println("findFreeItem");
  if(debug) System.out.println("LL : "+ll.size());
  Cacheable entry = null;
  LRUItem tmp = null;
  LRUItem element = null;
```

```
synchronized (this) {

  /// Traverse the list. Find the last item that can be
      reclaimed.

      for (int f=ll.size()-1;f>=0;f--) {
        tmp = (LRUItem) ll.get(f);
  entry = tmp.getEntry();
  if(tmp.isRemoveRequested()) continue;
  if(tmp.getKeepCount() == 0){

    if(entry==null||entry.getIdentity()==null){
            element=tmp;
            break;
          }

    if(entry.isDirty()) entry.clean(false);

    Object o = hm.remove(entry.getIdentity());


    if(SanityManager.DEBUG){

      SanityManager.ASSERT(o!=null,"Error in findFreeItem,
          no hash-entry");
      if(log){
        try {
          //lruDump.write(" evicted:");
          //lineNr++;
          PageKey pk = (PageKey)entry.getIdentity();
          long containerId = pk==null ? -1 : pk.
              getContainerId().getContainerId();
          long pageNumber = pk==null ? -1 : pk.getPageNumber
              ();

          lru2.write(" evicted:("+containerId+","+pageNumber
              +")");
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }

      //  entry = clearIdentity(entry);
    element = tmp;
    break;
  }
  element = null;
```

```
tmp = null;

    }

  // Did we find an available item?? If not go to list
     extension.
  if(element!=null){

    //  If list is longer than maxValue (someone extended
    //  the list because there were no unkept items)
    //  we will try to shorten the list.

    // How many items will we try to remove :

    int toLongby = ll.size()-maxSize ;

    tmp = null;
    if(toLongby>0){

      for(int i=ll.size()-1;i>=0;i--){

        if(toLongby==0) break;

        tmp = (LRUItem)ll.get(i);
        if(tmp==element) continue;
        if(tmp.isRemoveRequested()) continue;

        entry=tmp.getEntry();
        if(entry!=null){
          if(entry.getIdentity()!=null){
            if(tmp.getKeepCount()==0){
              Object o = hm.remove(entry.getIdentity());


              if(SanityManager.DEBUG)
              SanityManager.ASSERT(o!=null,"Error in
                  findFreeItem, no hash-entry");

            if(entry.isDirty())entry.clean(false);
            entry = clearIdentity(entry);
            }
            else continue;
          }

          // If the list we are working with is aIn, we
             must put the removed element in aOut.
          // If the list is aM, just forget the last item
          if(overflow!=null){
            overflow.addFirst(entry.getIdentity());
```

```
                    OverflowHash.put(entry.getIdentity(),entry.
                        getIdentity());
                    if(overflow.size()>overFlowSize){
                      Cacheable rmv = (Cacheable)overflow.
                          removeLast();
                      if(rmv.getIdentity()!=null)
                        OverflowHash.remove(rmv.getIdentity());
                    }

                  }

                }

                ll.remove(tmp);
                tmp = null;
                toLongby--;
              }
            }
            // return the item
            return element;
          }

          // There are no free items in the list. We have to
              extend the list by one.
          if (debug) System.out.println("No free item found, 
              extending the list");
          element = new LRUItem();
          ll.add(element);
        return element;
        }
}
```

4.4 List lookups and hash tables

All lists in LRU and LRU/2Q as described in this chapter have corresponding hash tables for fast lookup. Like for LRU, all lists have HashMaps containing references to the active LRUItems in the list. For larger caches the time saved by looking up a page in a hash will be considerable. Of course this also means extra complexity in the cache manager.

4.5 Build Derby to use a specific page cache manager

To build Derby to use a page cache manager of your choice only one line of code has to be added. The method newCacheMangaer of ClockFactory has to be altered like shown in this code example.

```
public CacheManager newCacheManager(CacheableFactory holderFactory
    , String name, int initialSize, int maximumSize)
{
  if (initialSize <= 0)
        initialSize = 1;

  //By adding this line a page cache manager named LRU2Q would be
     instanciated.

  if(name.equals("PageCache")) return new LRU2Q(holderFactory,
     name, initialSize, maximumSize, false);

  return new Clock(holderFactory, name, initialSize, maximumSize,
     false);
}
```

4.6 LRU

4.6.1 A Basic LRU

A basic LRU implementation is shown in this pseduo code.

```
BasicLRU{

ll = new linked list
hm = new HashMap

method find(key)
    item = hm.get(key)
    if item == null
        item = new Item
        hm.add(key,item)
        ll.removeLast()

    else ll.remove(item)
    ll.addFirst(item)

    return item
}
```

In a truly basic LRU implementation one would not need the hashmap. But including a hashmap increases performance on lookups dramatically. Traversing the linked list is expensive operation compared to hash lookup.

For LRU there are two main cases for cache finding an item in cache, hit or miss. A page miss cause a page to be faulted in and the last inactive page to be faulted out. This can be seen in figure 4.4. A page hit have the page move to the front of the LRU list as seen in figure4.3.

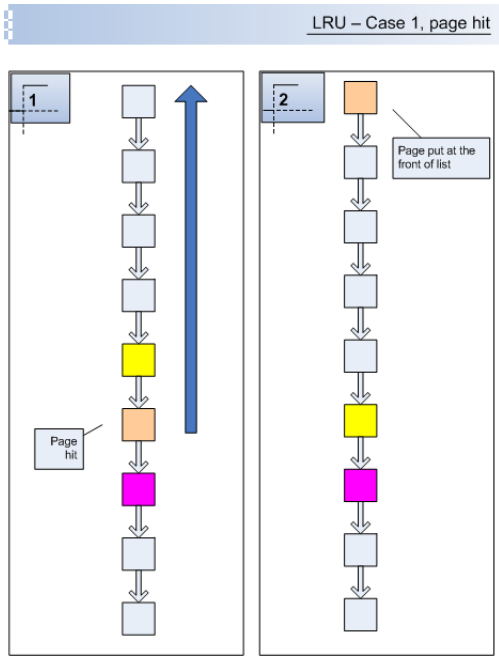

Figure 4.3: LRU page hit

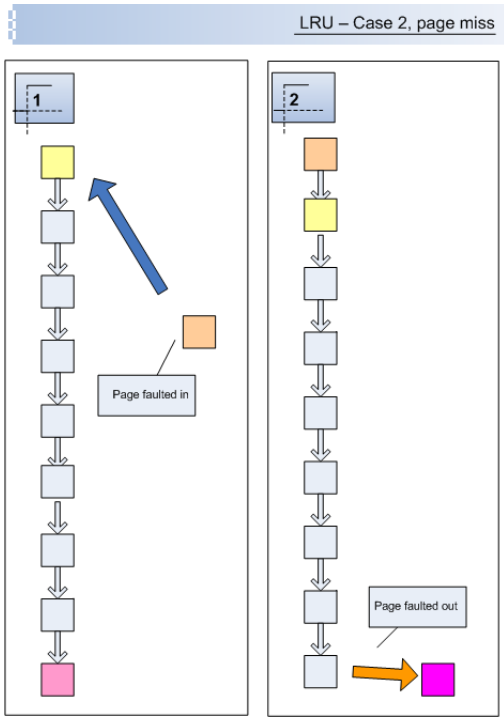

Figure 4.4: LRU page miss

4.6.2 The LRU implemention

Classes

Two classes will be implemented. The most important is the cache manager implementation called LRU. In additon to the cache manager we need to create the LRUItem class as described in section 4.3. Central classes for the implementation are show in figure 4.5.

Methods

The most important methods of LRU are find, release and findFreeIem. FindfreeItem was described in section 4.3. We will take a closer look at find and release.

Find

The find method is in interface method required for a cache manager. When Derby needs a cached item it calls this method. It contains the LRU logic as well as the mechanisms required by derby.

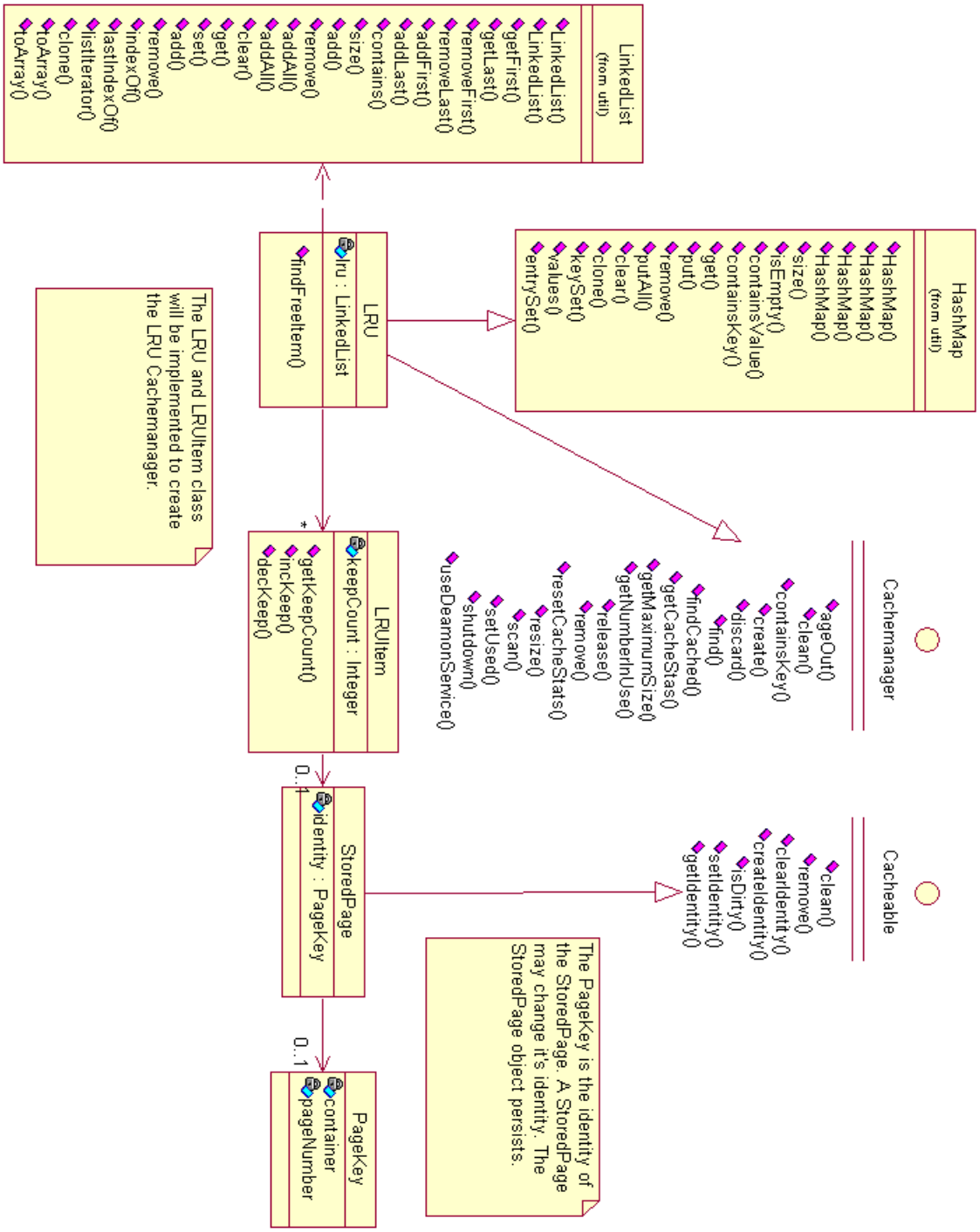

Figure 4.5: The most important classes of LRU caching

```
public Cacheable find(Object key)
        throws StandardException
    {
      if(debug)System.out.println("find");
      if(debug)nrFind++;
      Cacheable entry;
      LRUItem item;

      synchronized (this) {

      // If Derby in shutdown return null
           if(!active)
                return null;

           item = (LRUItem)get(key);

           if((item!=null)&&(item.getEntry()!=null)){
             entry=item.getEntry();
      }

           if(item == null)
           {
             /// CASE 2 - Item not found in cache
             stat.findMiss++;
             item = findFreeItem(lru,(int)MAX_ENTRIES,this);
               entry=item.getEntry();
               entry =setIdentity(entry,key);
               item.setEntry(entry);
               lru.remove(item);
               lru.addFirst(item);
               item.incKeep();
               put(key, item);
           }
           else{
             // CASE 1 - Item found
             stat.findHit++;
             entry = item.getEntry();
             lru.remove(item);
             lru.addFirst(item);
             item.incKeep();
           }
      }
      if(SanityManager.DEBUG) return verifyFind(entry,item,key);
      else return entry;
    }
```

Release

Release is an interface method for Cache manager. It is called imediatly after a Derby is finished with the page reference. After a page has been released the caller must throw away the reference to the page. The way find and release interacts can be described in this pseudocode. The java code shows the release impementation for LRU. Note that it is in release that the keepCount an item is decreased.

```
codeSegment{
page = CacheManager.find(PageKey)
page.doSomeOperation()
CacheManager.release(PageKey)
page=null
}
```

```
public void release(Cacheable entry)
    {
      if(debug)System.out.println("release");
        LRUItem item = (LRUItem)get(entry.getIdentity());

        int keepCount = item.getKeepCount();
        ///
        synchronized(this)
        {
          item.decKeep();
          if(SanityManager.DEBUG){

            if(debug)verifyRelease(entry,item,keepCount);
          }
        }
    }
```

4.7 LRU/2Q

LRU/2 has a more agressive rule for allowing items to the cache. Different implementations exist but they all have in common that items accessed two times have priority over items only accessed once. The two types of items are called respectivly hot and cold pages. Original LRU/2 used a priority que to distinguish between hot and cold pages. Maintaining a priority que has logaritmic complexity, and so this form of LRU/2 has a very large processing overhead.

LRU/2Q is a more effective implementation with only linear complexity. LRU/2Q maintains 2 lists (2Q). One for single accessed pages or co-related references. The other for hot pages. In addition to the two buffers the algorithm keeps a list over page identifiers over recently used pages. By keeping only the identifiers and not the pages themselves page history can be remembered at a low cost.

4.7.1 Description of LRU/2Q

Figure 4.6 gives a graphical description of the algorithm. The three structures of LRU/2Q are called

- aIn - The first que is FIFO managed. Its purpouse is to hold a given page for the period it is accessed in co-related references. This is a type of access where a page is referenced many times during a short period and then left alone. If LRU/2Q is well tuned this buffer, such accesses will not reach the main buffer AM

- aOut - This is a FIFO que holding pointers to pages thrown out of aIn. Because it holds pointers, and not the pages themselves, it can keep a lot of items at a low price. If a page is re-referenced for the period it is remembered in aOut it is probably a hot page and is admitted to the main buffer.

- AM - AM is an ordinary LRU list. Pages come from aOut. Pages are screened in aIn before they are allowed in so the buffer will hold mostly hot pages. When the buffer is full pages are faulted out of cache from the last position in the list.

Page requests to LRU/2Q can be divided into 4 cases.

- CASE 1 - item found in aIn
- CASE 2 - item found in aOut
- CASE 3 - item found in aM

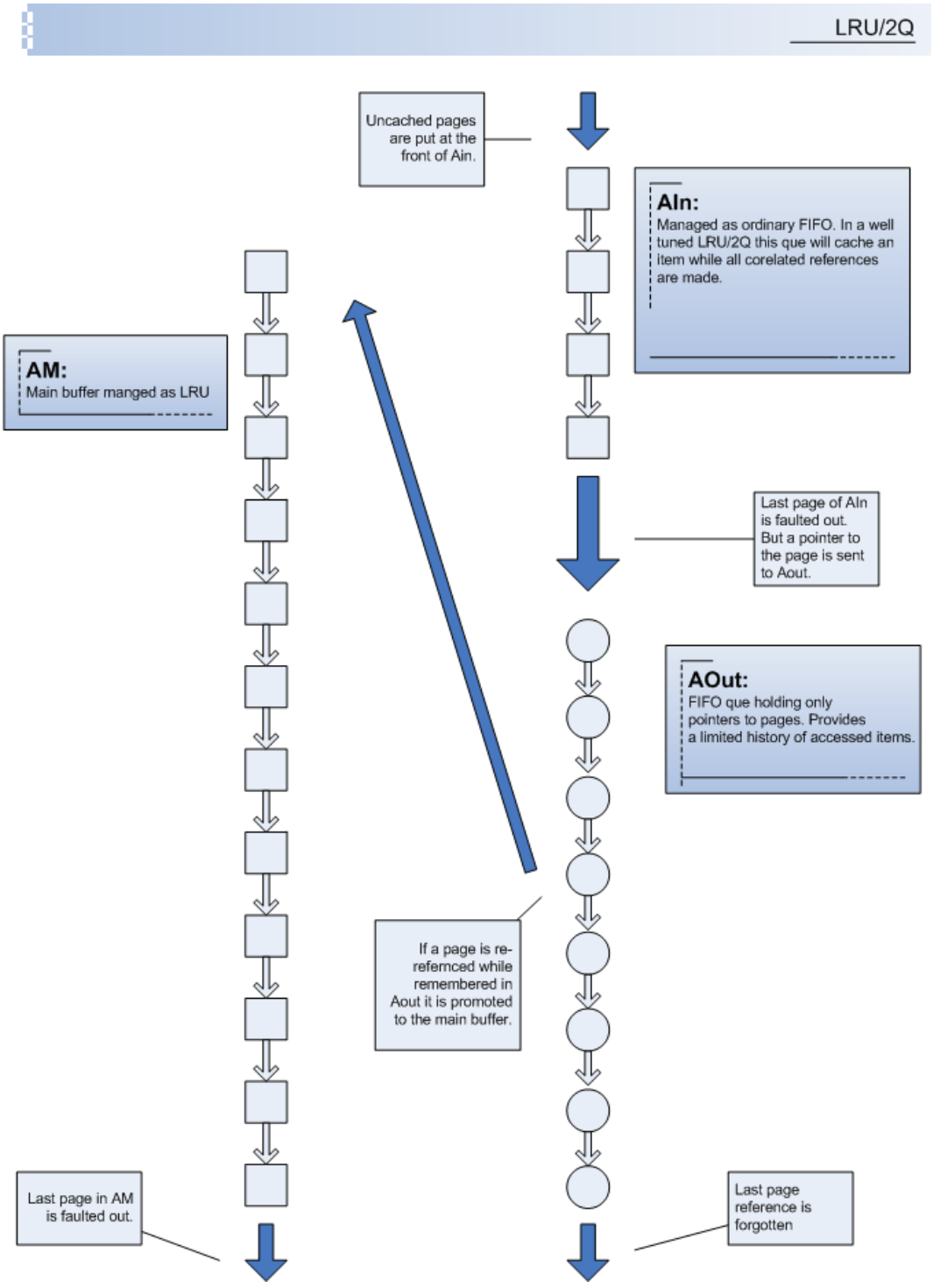

Figure 4.6: Structures and most important opertions of LRU/2Q

- CASE 4 - item not found in cache

We will give a complete study of each of the cases.

CASE 1 - Item found in aIn

Figure 4.7 shows the aIn list. If a page is found in aIn the cache remains exactly the same. No items are faulted in or out of cache. aIn is FIFO managed so the items remain in same place. The purpose of the aIn list is to hold pages for the period they have co-related references. This allows some screening before promoting items to main cache (aM), while still giving page hits for co-related references.

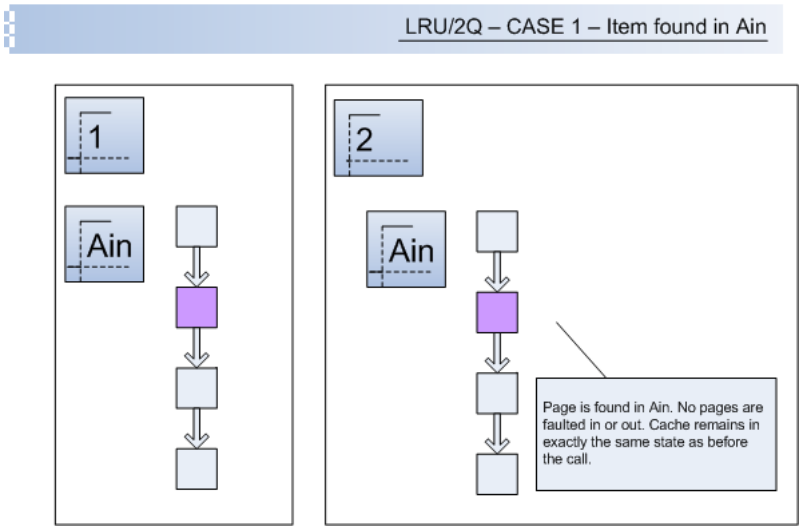

Figure 4.7: Item found in aIn

CASE 2 - Item found in aOut

Figure 4.8 shows the aOut and Am ques. Note that aOut contains page references, not pages themselves. Hits in aOut counts as page faults in statistics. When a an item is found in aOut the page is faulted in and put at the front of Am. The last page of Am is faulted out.

CASE 3 - Item found in aM

Figure 4.9 shows the aM que. When aM takes a page hit, it acts as an ordinary LRU hit. No pages are faulted in or out.

CASE 4 - Item not found in cache

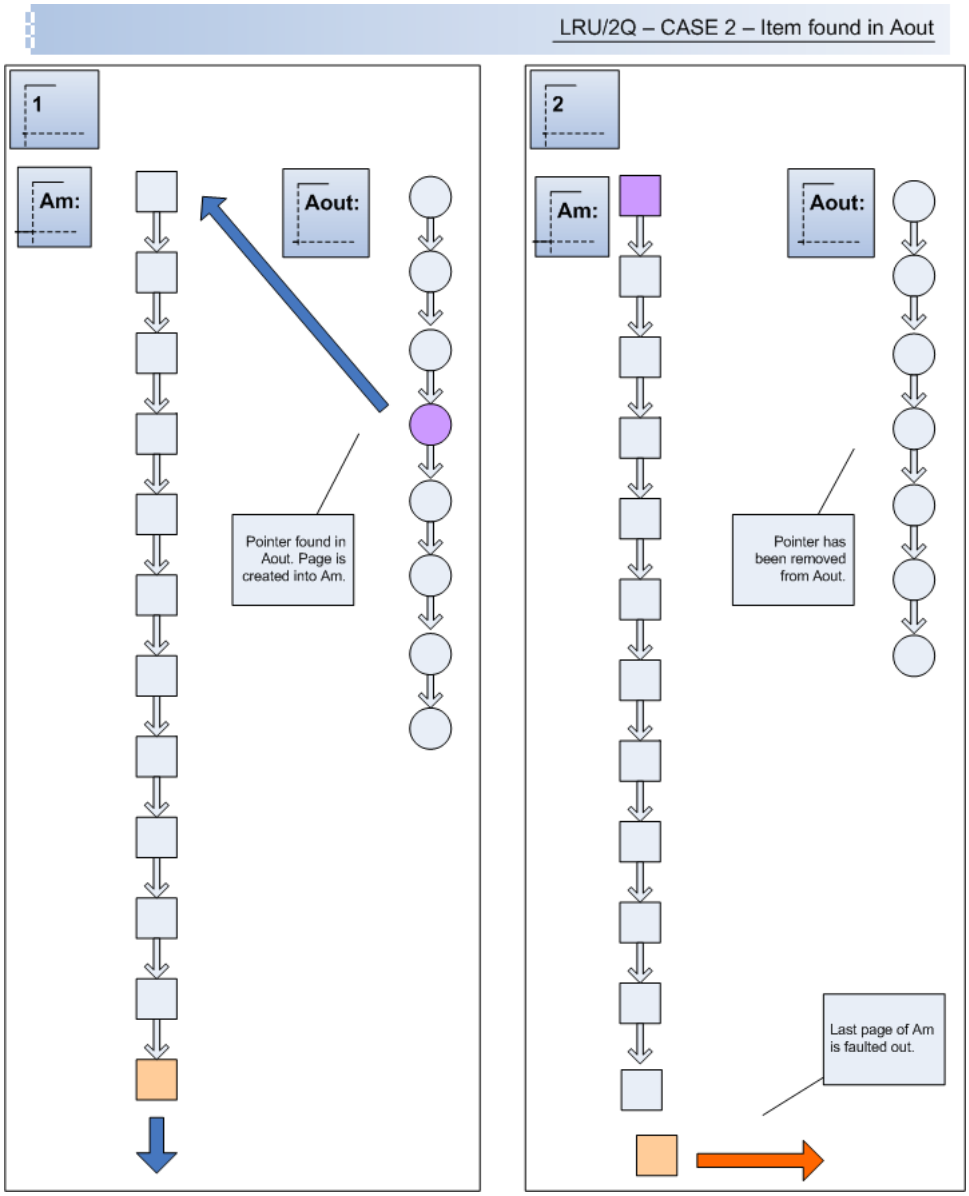

Figure 4.8: Item found in Aout

Figure 4.9 shows the aIn and aOut ques. The page is not found in cache. It is faulted in and put in aIn. The last item of aIn is faulted out, and the page key is sent to aOut. The last item of aOut is removed and forgotten.

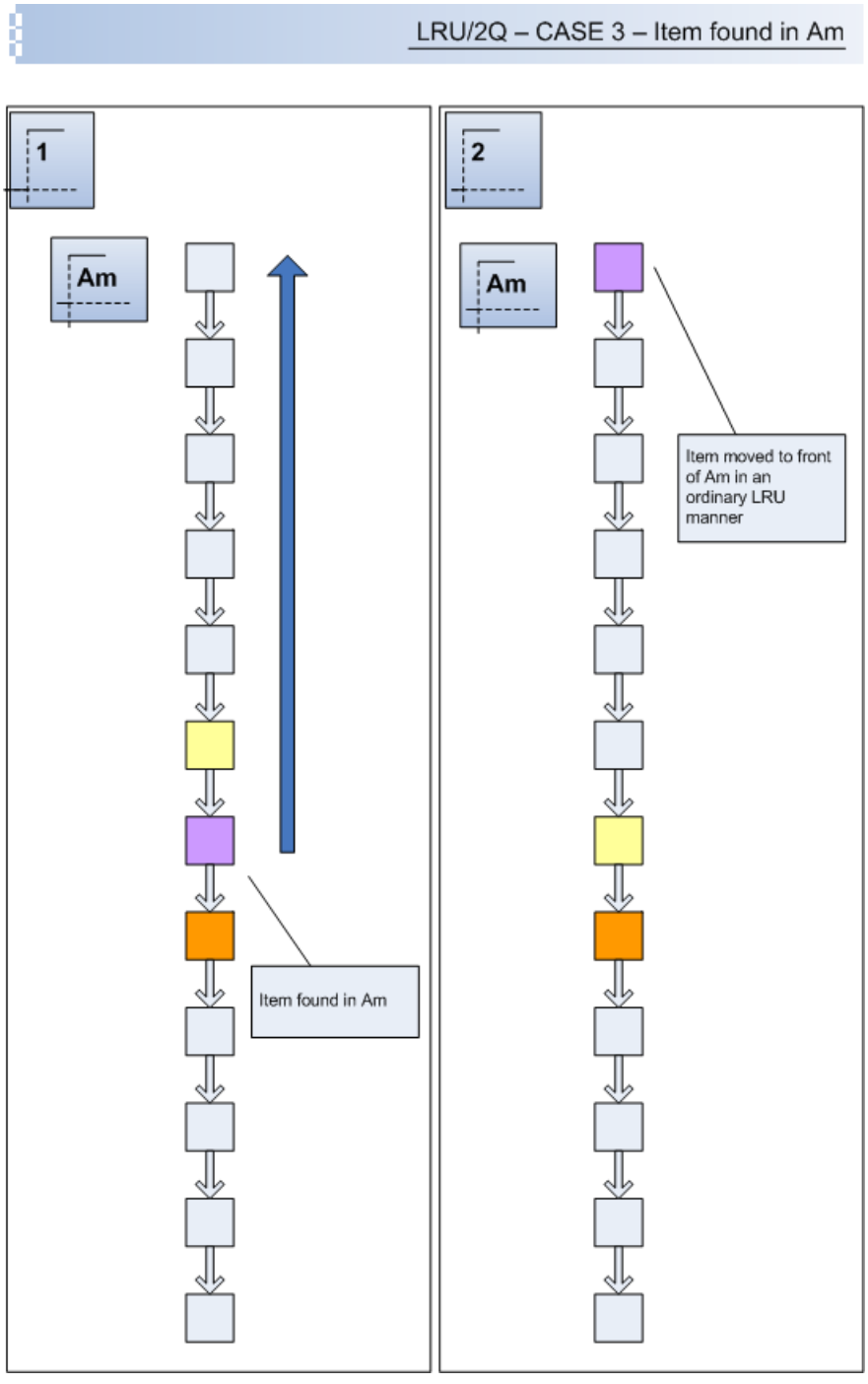

Figure 4.9: Item found in Am

4.7.2 Implementing LRU/2Q

Much of the ground for implementing a CacheManager has already been covered by the LRU implementation. But there are major differences in datastructures and the

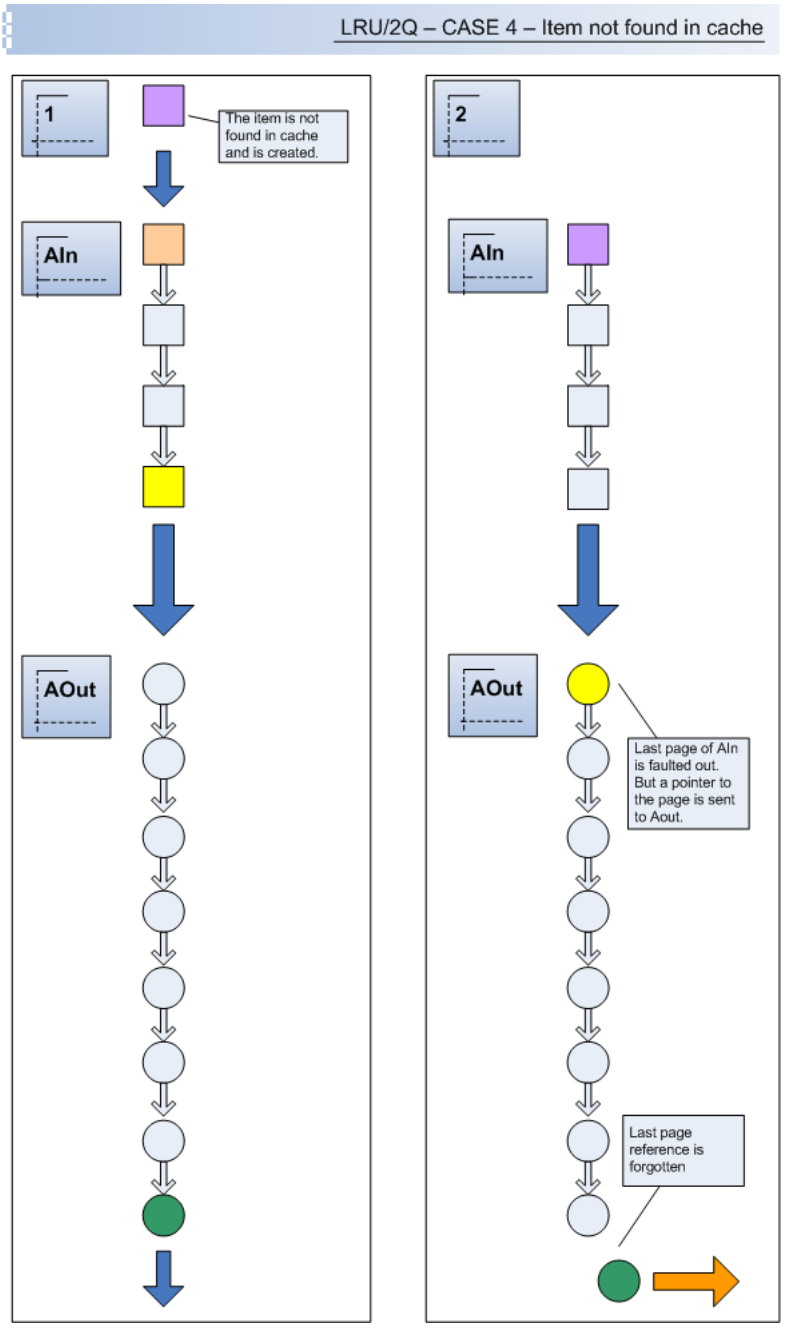

Figure 4.10: Item not in cache

logic maintaining them.

Classes

The classes for LRU/2Q are mostly the same as for LRU. The changes are found

in the new CacheManager implementation. The class diagram is found in figure 4.11

Methods

The most important methods of LRU/2Q are find, release and findfreeItem. The later is the same as for LRU, as described in section 4.3.

Find

The find method will have reflect the changes in from LRU. This results in a total rewrite. The four cases discussed in 4.7.1 can be found in the code.

```
public Cacheable find(Object key)
    throws StandardException
{
  if(debug) System.out.println("find");
  nrFind++;
  Cacheable entry;
  LRUItem item;
  // If Derby is in shutdown return null
  if(!active)
        return null;

    /// CASE 3 Find page in AM.
  /// Move page to front of AM.

    item = (LRUItem)aMHash.get(key);
    if(item != null) {
      //finnes i aM
      aM.remove(item);
      aM.addFirst(item);
      entry = item.getEntry();
      item.incKeep();
      stat.findHit++;
      return verifyFind(item.getEntry(),item,key,aM,aMHash);
    }

    /// CASE 2 Page found in aOut
    /// Page faulted into Am


    PageKey pk = (PageKey)aOutHash.get(key);
    if(pk != null) {
      item = findFreeItem(aM,kM,aMHash);
      entry = item.getEntry();
      if(entry==null)
        entry = holderFactory.newCacheable(this);
      else if(entry.getIdentity()!=null)entry=clearIdentity(
          entry);
```

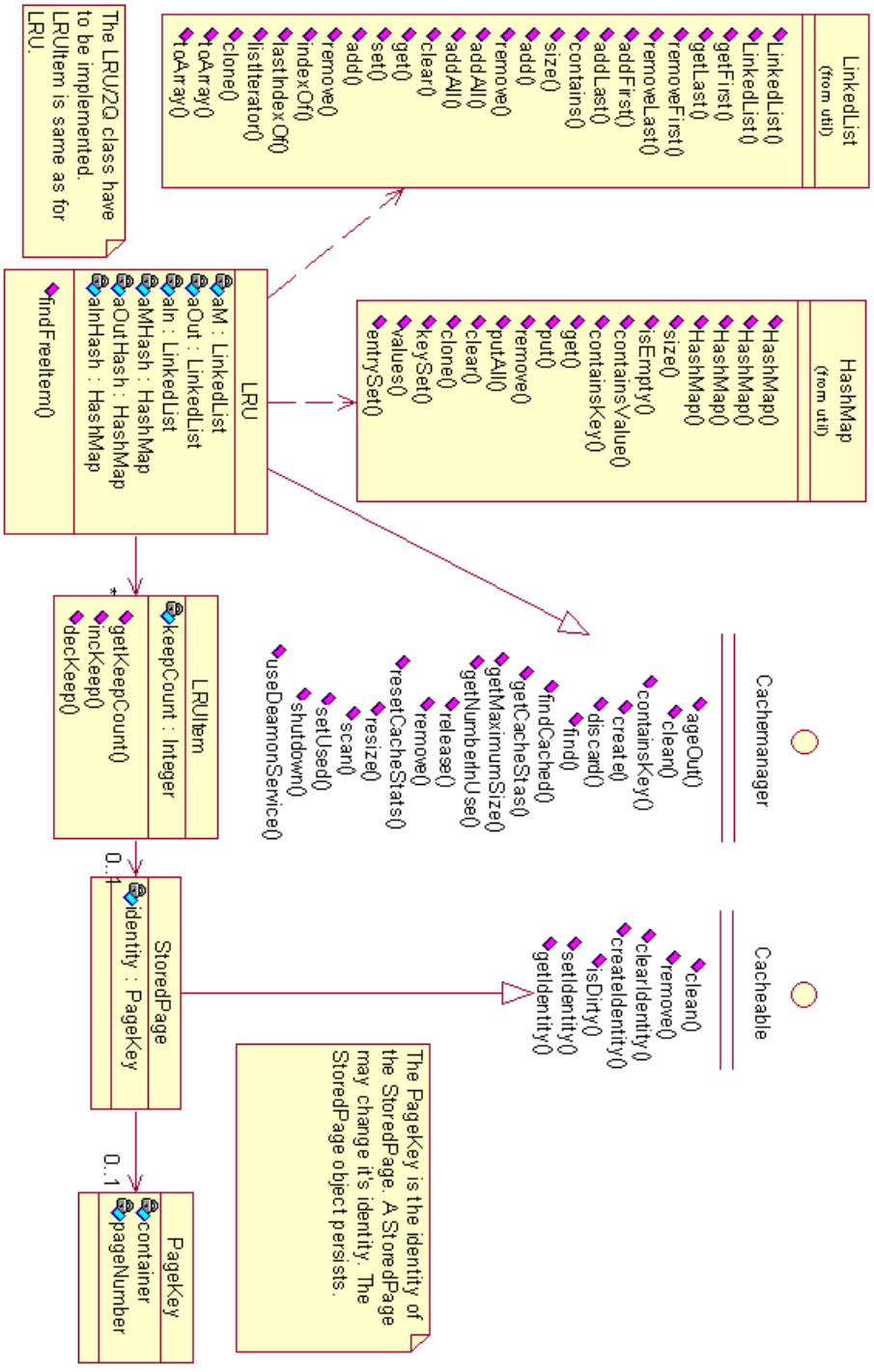

Figure 4.11: LRU2Q

```
    entry = setIdentity(entry,key);
    item.setEntry(entry);
    item.incKeep();

    aM.remove(item);
    aM.addFirst(item);
    aOutHash.remove(key);
    aMHash.put(key,item);
    aOut.remove(key);
    stat.findMiss++;
    return verifyFind(entry,item,key,aM,aMHash);
  }
  // Case 1 - Item found in aIn, do nothing
  item = (LRUItem)aInHash.get(key);
  if(item != null) {
    item.incKeep();
    stat.findHit++;
    return item.getEntry();
  }

  // CASE 4 - Item not found in cache.
  // Fault in and put in front of aIn.
  item = findFreeItem(aIn,kIn,aInHash);

  entry = item.getEntry();
  PageKey toOut;
  if(entry!=null){
    if(entry.getIdentity()!=null){
      toOut = (PageKey)entry.getIdentity();
      aOut.addFirst(toOut);

      if(aOut.size()>kOut)
        aOutHash.remove((PageKey)aOut.removeLast());
      aOutHash.put(toOut,toOut);

    }
    entry = clearIdentity(entry);
  } else {
    entry = holderFactory.newCacheable(this);
  }

  entry =setIdentity(entry,key);

  aInHash.put(key,item);
  aIn.remove(item);
  aIn.addFirst(item);

  stat.findMiss++;
```

```
        item.incKeep();
        item.setEntry(entry);

        if(debug) return verifyFind(entry,item,key,aIn,aInHash);
        return entry;
        }
    }
```

Release

The release method of LRU/2Q have to search both aIn and AM to find the page to be released.

```
    public void release(Cacheable entry)
    {
      if(debug) System.out.println("release");
      synchronized(this)
        {
          // See if page is in aIn
        LRUItem item = (LRUItem)aInHash.get(entry.getIdentity());

        // If page is not in aIn it must be in Am.
        if( item == null)
            item = (LRUItem)aMHash.get(entry.getIdentity());

          int keepCount = item.getKeepCount();

          item.decKeep();
        if(SanityManager.DEBUG){
          SanityManager.ASSERT(!((aInHash.get(entry.getIdentity())
              !=null)
              &&(aMHash.get(entry.getIdentity())!=null)),"Error, "
                  item in both aIn and aOut");

          if(debug)verifyRelease(entry,item,keepCount);
        }
        }

    }
```

Chapter 5

Validation

5.1 Introduction

Two aspects of correctness for the implemented algorithms will be tested.

1. Derby Compatibility - Correct behavior in respect to the Derby code. This basically means running without crashing or hanging.

2. Algorithm Correctness - Correct behavior according the algorithm specifications. Management of the cache structures must be correct according to the algorithms and the constraints set by derby.

The testing should put the system through heavy load conditions to expose as many errors as possible. The testing described in 1 are largely taken care of by Derbys SanityManager. Derby code running in debug mode is very defensive, checking many kinds of values. Examples of tests are

- is a page latched, and is it supposed to be latched?

- do log numbers correspond with page log sequence number?

Any abnormal behavior and Derby throws an exception and shuts down the database. This testing will therefore run in the background during all other testing, and running Derby in general. Heavy load is required, especially to finding synchronization errors.

The tests described in 2 will be tested individually for each implementation. This is described in sections 5.3 and 5.4. Note that these chapters only show examples how the data that can be retrieved from the logs. Analysis of logs for real data volumes require spreadsheets with a scale of (hundreds X tens of thousands). Section 5.2 describes the logging system.

5.2 Logging cache state

The logging system records the state of the cache. This is implemented in both LRU and LRU/2Q. The system will allow the behaviour of the algorithms to be studied. This is needed for validation, but is also useful for performance analysis. For every find called on the Cache manager a line is printed to the log file. The line contains the content for each of the lists found in the respective cache manager. A line from LRU can look like this:

line:65 —(209,1)—(113,1)—(192,1)—(113,0)—(96,1)—(96,0)—(209,0)—(192,0)—(16,1)—(16,0)—(-1,-1)—....—(-1,-1) find::(209,1)

Each page is identified by a container id and a page number. The page (209,1) has container id 209 and page number 1. Entries (-1,-1) represent empty LRUItems.

Number of finds can quickly come up to tens or hundreds of thousands. To be of any value for such large numbers of records the log must be imported into a spreadsheet. In the spreadsheet a given pages position can be tracked by, for instance, colors. Figure 5.1 shows an example of log for LRU/2Q. The leftmost table shows aIn, the center table shows aOut and the right table show aM. The column to the extreme right shows which page is being request before the line is written.

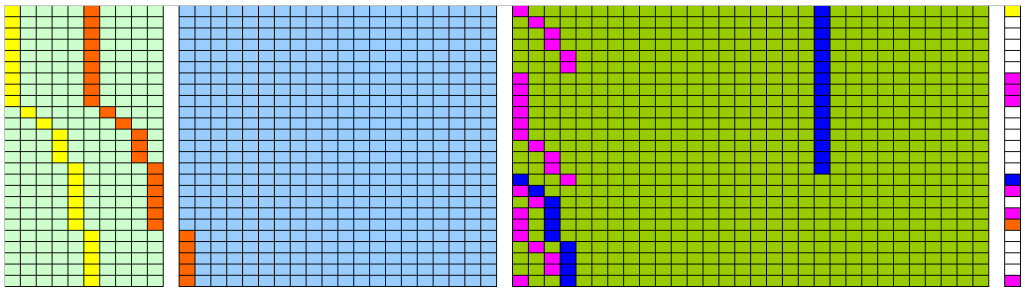

Figure 5.1: Example of cache log. LRU/2Q shown

All the testing in this chapter is with a page cache size of 40, which is the smallest size allowed in Derby. Keeping the cache size small makes it easier to analyze the results. The correctness of the algorithms should not be affected by the cache size.

Note that if an item in the aOut list of LRU/2Q appears to advance one position in the que, it is because items can be removed from the middle of aOut.

5.3 LRU

5.3.1 Algorithm Correctness

LRU will be tested in respect to the two cases represented in figures 4.4 and 4.3, cache hits and cache misses. The log is studied inside a the spreadsheet. Random segments

of log from both initialization and normal operation. Only one demonstration of each case is presented here.

Case 1 - cache hit

Figure 5.2 shows example of the log for a page hit. The page is found in the buffer and moved to the front of the que.

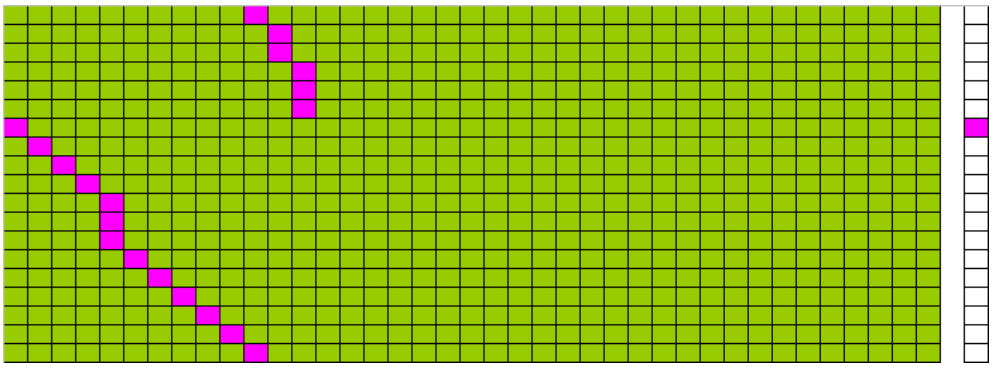

Figure 5.2: LRU cache hit

Case 2 - cache miss

Figure 5.3 example of the log for a page miss. The page is not found in the buffer. The page is faulted in and put at the front of the que.

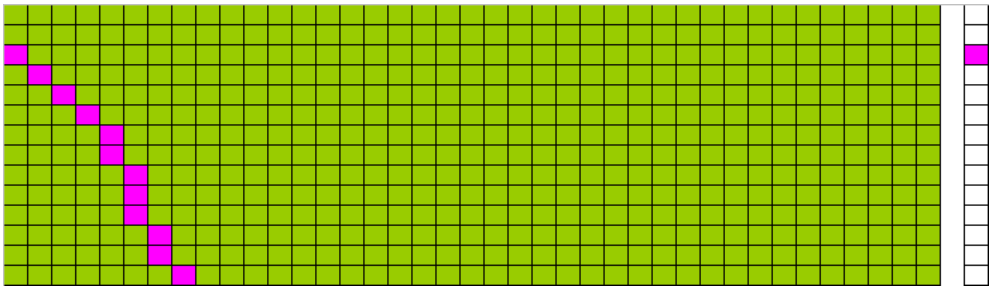

Figure 5.3: LRU cache miss

5.4 LRU/2Q

5.4.1 Algorithm Correctness

LRU/2Q will be tested with respect to the four cases as described in section 4.7.1.

Case 1 - Item found in aIn

Figure 5.4 shows an example of the log when a page is found in aIn (the second shown hit). On the page hit nothing happens.

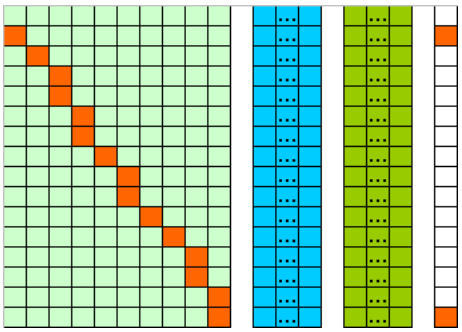

Figure 5.4: LRU/2Q - Case 1

Case 2 - Item found in aOut

Figure 5.5 shows an example where a pagekey is found in aOut. The page is faulted in and put at the front of aM.

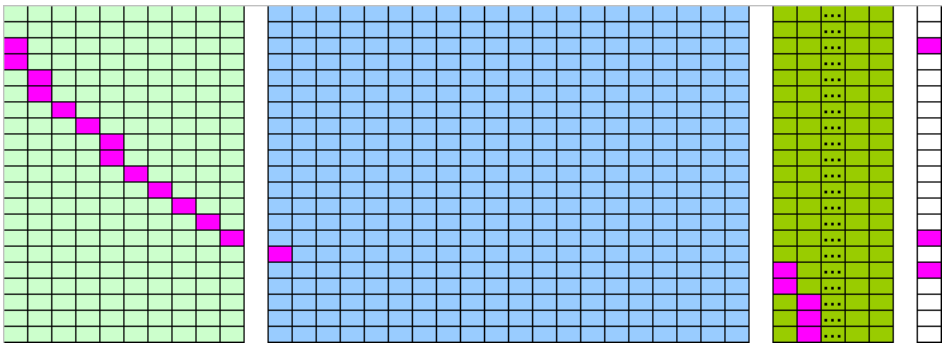

Figure 5.5: LRU/2Q - Case 2

Case 3 - Item found in AM

Figure 5.6 shows an example where the page is found in Am. The page is moved to the front of the que.

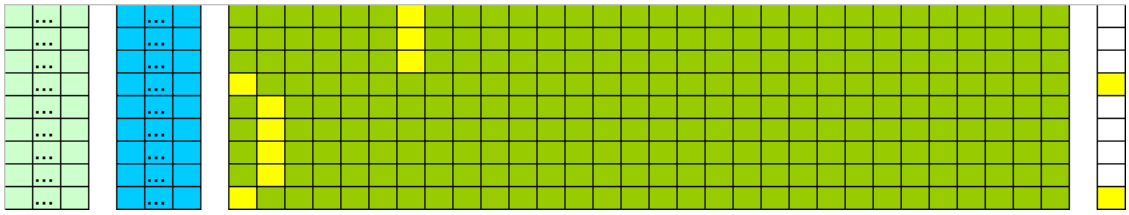

Figure 5.6: LRU/2Q - Case 3

Case 4 - Item not found in cache

Figure 5.7 shows an example where the page is not found in cache. The page is faulted in and put at the front of aIn.

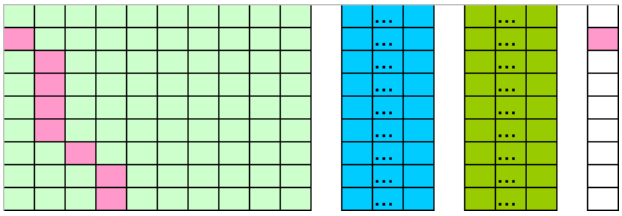

Figure 5.7: LRU/2Q - Case 4

Chapter 6

Testing

6.1 Introduction

There are many different parameters to take care of when testing a database system cache. These include database size, total cache size, number of tables, patterns of queries, page size, tuple size and many more. These parameters will heavily influence the performance of a page replacement algorithm. Instead of tweaking tests to be fully optimized for one, we will in this chapter do several tests in order to find strengths and weaknesses to the three algorithms available, clock, LRU and LRU-2Q.

It is more or less impossible to know what a typical database is, and what kind of traffic that will run on it. Some databases are very static, and is used mostly for lookups, while others are more dynamic, and changes the content a lot. Because of this it is hard to test the real performance of a cache algorithm, since it will vary a lot for each application of the database. There are some standard tests, like TPC and others. These does not directly test the performance of the page replacement algorithm.

6.1.1 Measuring units

It is basically two measuring units for testing the performance of a page replacement algorithm. These are

- Cache hit rate
- Time

These two are often correlated, since in the case of a page miss, the time to fetch the page from disk, is usually much greater than the extra overhead one algorithm has over another. We will, however, focus our testing here on hit rate. This is because both our extra implementations should be considered as prototypes. It is most likely that they can be tuned to be faster since we have been very conservative with the synchronization. It is also possible that additional functionality must be

added before it can be put in production. Such functionality might be for example debugging code.

We will however show a time experiment at the end of this chapter.

6.2 The tests

In our tests we use small cache sizes and slightly bigger tables. We set the pagesize to the minimum allowed by Derby, 4096 bytes. The minimum number of cache size allowed is 40 pages. Unless otherwise stated, all our tests run 20 simultaneous threads, to simulate a multiuser environment. We create a test set of queries for each thread to execute. The sets are created using random functions, but the same sets are run on each of the three algorithms. Each of the threads will execute 200 different queries. We mix the queries between ordinary lookups and scans. There is a 5% chance that each query will be a scan. We run each test several times(6-10), to get an average dataset. This is because the variation in the tests is quite big, and the graphs looks smoother with the average of several tests. Since all tests are run at random, it is impossible for us to tweak the tests to perform better on one algorithm than the other.

We will use two different test tables. The first is a table where each tuple fill one page. We do that by having one identificator field, which is the primary key. The other field is a dummy varchar field of 3500 characters. We refer to this table as Table1.

The second is mostly the same, but we adjust it so that each page contains ten tuples. This is to see if there are any difference in performance for the two cases.

6.2.1 Test 1: 80/20 distribution

We start with a test where 80% of the lookups is done on 20% of the data. We does this at random, by first picking which pages to be in the 20% hot set. The test is run where the cache with different sizes of the cache on all the algorithms. Table1 is populated with 400 tuples, so that the database uses 400 pages. Hence the total database size is approximately 1600Kb. We vary the cache size from containing 40, 80, 120 to 400 pages. Hence it is going from 10%, 20% and up to 100% of the total database size. The results are shown in figure 6.1.

There seems to be just a minimal difference between LRU and LRU-2Q, but they both outperform the clock in this test. LRU-2Q beats LRU in by a percent or so. It is also worth to notify that while the LRU algorithms increase the hit rate linear as the cache size grows, the clock is struggling to take advantage of higher cache sizes.

We were somewhat surprised that the hit rate never went to 100% , but it is simply because other information, such as the indices also are stored in the pages. Another reason, is that filling the cache at startup causes several cache misses, but this will not make too much significance in the final results.

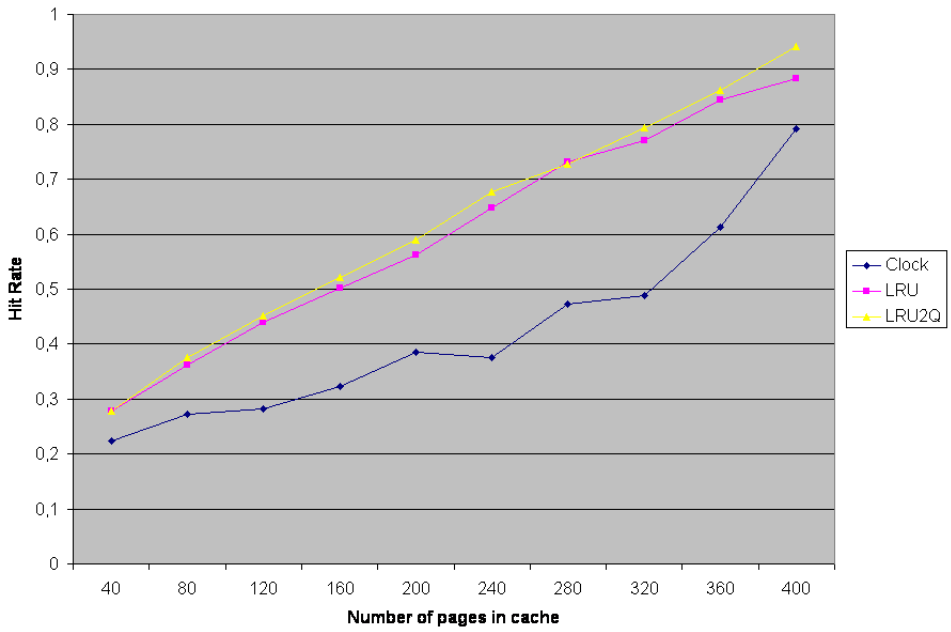

Figure 6.1: Test 1. 80/20 distribution with one tuple per page

6.2.2 Test 2: 80/20 distribution, 10 tuples per page

This is basically the same test as the previous one, but we change to Table2, and with 4000 tuples. The Size of the database remains the same. Again the tests are run, varying the page sizes. The results are shown in figure 6.2

The results are not too different from the previous one, but there are less difference between the algorithms' performance. This may be explained by that the 20% popular pages are selected at random, and there are probably more pages in the hot set, because a page in this case will consist of several tuples.

A second thing worth to notify, is that the LRU-2Q is performing worse than the other when there is a small page cache, 10%. An investigation of this revealed that the combination of page size, record size and cache size created problems for LRU/2Q. A look at the log, as shown in figure 6.3, shows why it performed so badly. The access pattern displayed in the figure was repeated all through the test. The pattern consisted of three accesses. By the last access the page had been moved to aOut. The page was then moved to aM. In this case the aIn que did not work as it is supposed to, as it did not screen aM from co-related references 3.1.4. By increasing the cache size over a certain threshold LRU/2Q would again perform well.

6.2.3 Test 3 and 4: Scan rate

In this section we try to change the rate of which a scan will occur. This is interesting, because one of the main arguments for using the LRU-2Q page replacement

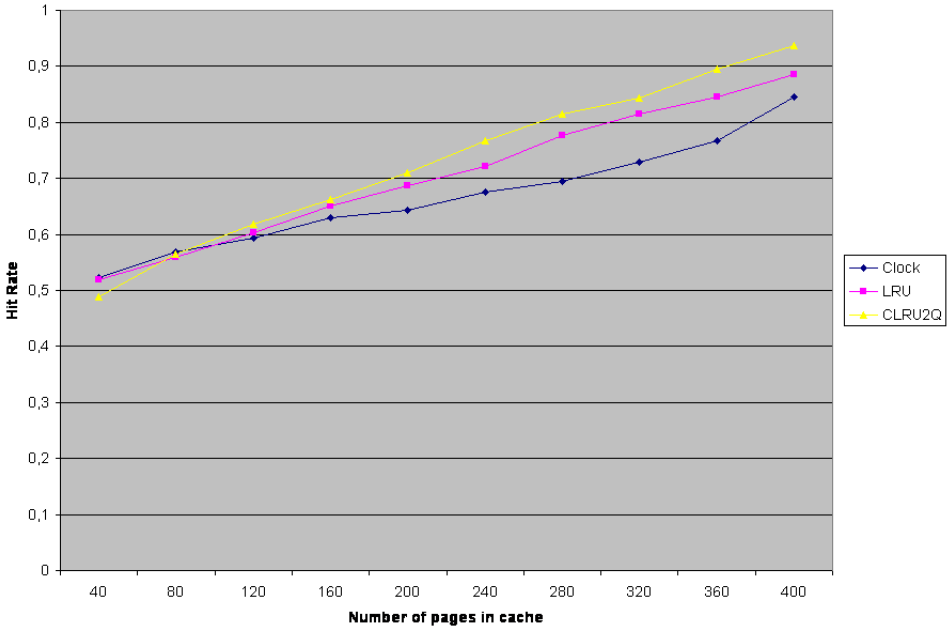

Figure 6.2: Test 2. 80/20 distribution with 10 tuples per page

algorithm is that it is scan resistant. The scan rate we explore closer are 0% , 1%, 5%, 10% and 20% . That is, how frequent a scan will occur in the queries we execute. We run two tests, both using Table1 and 400 tuples. In one of the tests we use a cache of 80 pages(20%), and the other with 160 pages(40%). The results are shown in figure 6.4 and 6.5. In these tests we also use the 80/20 distribution when doing lookups.

The difference of LRU-2Q versus clock is as expected. The LRU-2Q is far more resistant to scans, maintaining a good hit rate even when we increase the scan rate. What might be a little surprising is that the LRU performs almost as good as LRU-2Q.

6.2.4 Test 5: Even distribution

The next test is run with an even distribution, and not the 80/20. This means that all the lookups are done at completely random among the whole table. We continue to use Table1. Again we vary the page cache size from 10 to 100% of the table size, and measure the hit rate at the respective sizes. The scan rate is set to 5% . The results are shown in figure 6.6

Again the LRU-2Q shows the best performance, just better than LRU. The clock is again ending up last. The hit rate for clock is growing slower, except from when the cache size is closing up to 100% of the total table size. The results are not too different from the 80/20 test 6.1, but LRU-2Q is doing better than LRU here. That

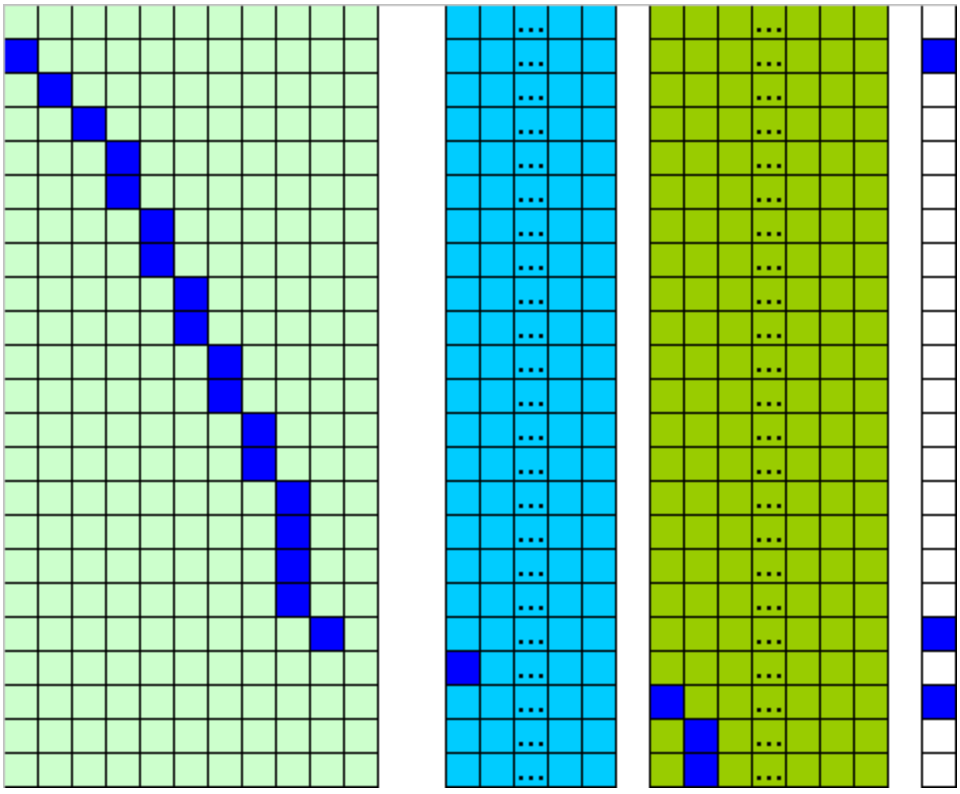

Figure 6.3: Three co-related references. aIn does not hold the page long enough

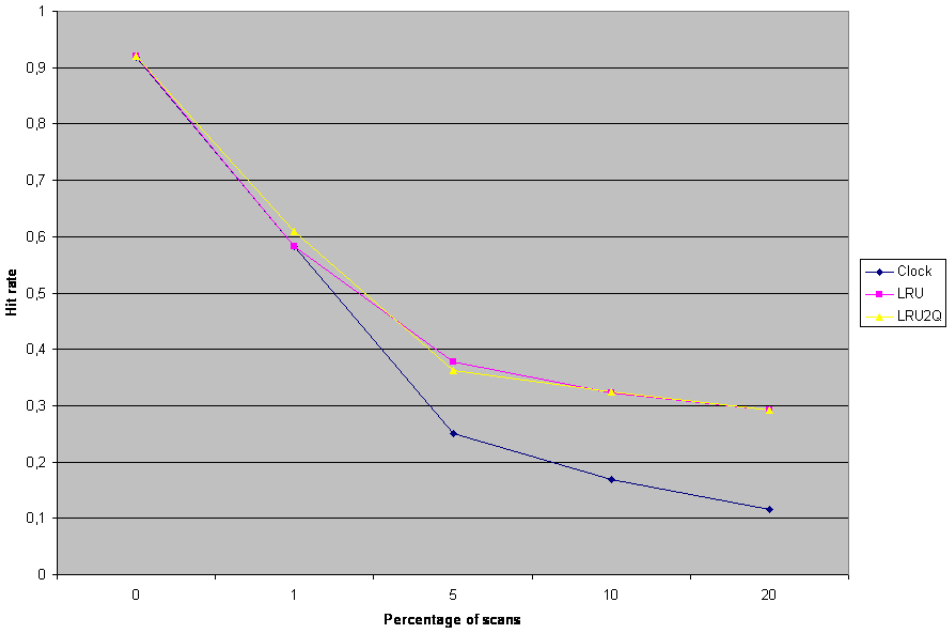

Figure 6.4: Test 3. Varying scan rate, page cache: 80 pages

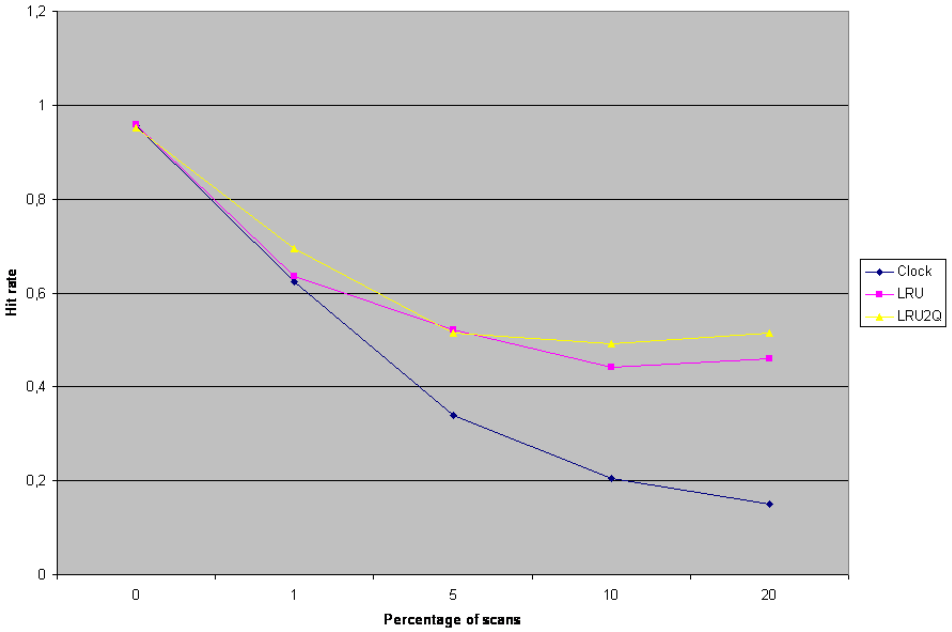

Figure 6.5: Test 4. Varying scan rate, page cache: 160 pages

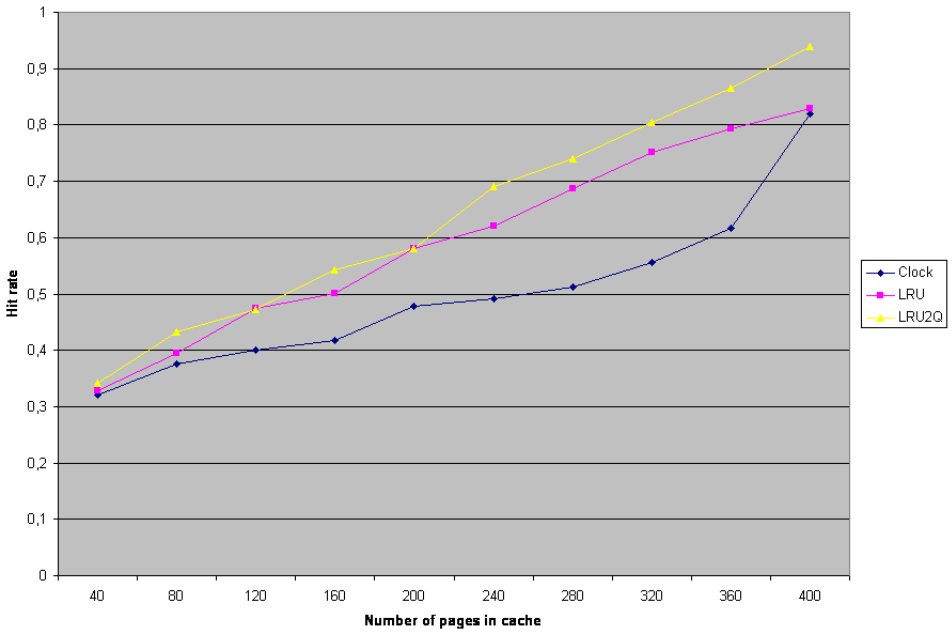

Figure 6.6: Test 5: Even distribution

Clock	14118,8	14308,55	14075,9
LRU	11682,1	10992,9	10932,8
LRU2Q	10746,15	11193	11251,4

Table 6.1: Average time in milliseconds

is because in the 80/20 test it is more likely for the LRU to hit the popular pages.

6.3 Time test

As we mentioned in the beginning of this chapter, the time tests should not retain too much focus, but we will anyway try to run one test to see which one is faster. The time is taken when each thread is starting, and stopped when it is finished. We then receive 20 numbers, and we present the results as both and average in table 6.3 and the maximum in table 6.3.

As we can see, the results are more or less the same as in the hit rate tests.

Clock	16485	16782	16766
LRU	13766	13313	13187
LRU2Q	13515	13953	13765

Table 6.2: Max time in milliseconds

The LRU and LRU-2Q are faster than the clock. It is, however important to again notice that some additional testing should be done before these algorithms should be considered as ready for a release.

The LRU again proves to be doing better than expected. In this case we will argue that the dataset is too small for the LRU-2Q to be performing at max. Since the complete database is at 1600Kb, the time to fetch a page is small, and the extra overhead for the LRU-2Q shows that it is just smaller in this case. They are, however, both faster than clock.

6.4 Test analysis and conclusions

One thing all the tests have in common is that both LRU and LRU/2Q have better hitrates. This is true for all combinations of page and cache sizes with the one exception mentioned in section 6.2.2. Section 6.3 suggests that they also are faster, although these tests are much less comprehensive and do not carry as much weight. LRU and LRU/2Q also seem to scale better than Clock. LRU and LRU/2Q seem to scale linearly with increasing cache size. But Clocks gain in performance is more irregular. With these results we can conclude that both LRU and LRU/2Q are an improvement over Clock. We would have expected LRU/2Q to outperform LRU. While LRU/2Q is better in most of the tests its advantage over LRU is not large. LRU also seem to scale as well as LRU/2Q.

Do the results justify changing the page Cache algorithm? The LRU/2Q algorithm in its present state is only a prototype. Many methods of the defining interfaces have not been implemented. Example of methods that remain unimplemented are all the methods of the Serviceable interface. These methods are not called during the relative short period the tests we have conducted take. But when running for longer periods, like days or weeks these methods probably will be called (We are not sure. They may only be in use for the other types of cache managers like, for instance, statement caching).

Another issue that should be addressed is the relative conservative synchronization used in LRU/2Q. During debugging of the class synchronization blocks were made larger than needed in order to track down bugs. These blocks could be reduced, and this would probably give slightly better performance. But this is very elaborate work, as the risk for introducing new synchronization errors is considerable. Certain synchronization errors have a way of not manifesting themselves, except for in extraordinary situations.

A complete study of page algorithms for Derby would include an implementation of CAR/CART. CAR/CART was the only real competitor when we chose algorithms to implement. CAR/CART employs some of the same strategies as LRU/2Q, like screening the main buffer from co-related references. Seeing how a CAR/CART implementation would have performed on the same tests would could prove a decisive point for the final choice of page cache algorithm.

To sum things up, LRU/2Q performs best of the three algorithms, closely followed by LRU. Different values for cache and record sizes give different hitrates. This suggests that LRU/2Q can be tuned (cachesize) to perform well on all the datasets we have tested. The question remains, if the gain in performance is big enough to justify changing the algorithm. Also a CART implementation should be made, and then CART would be tested against LRU/2Q.

Chapter 7

Summary

This study started by exploring the cache mechanism in Derby, and evaluate the possibilities to change the current page replacement algorithm with one more suitable to the database management paradigm.

We started by studying the theoretical background for page replacement algorithms. We continued by studying the Derby architecture, and getting to know how the caching was working. We found out that it was quite easy to replace the page caching function in theory. It was however more difficult to do the actual implementations. This is because the Derby system is not very well documented and there are many dependendcies inside the system. Eventually we managed to implement both the LRU and the LRU-2Q caching algorithms. Our implementation need to be furthered tested (and probably debugged) in order for it to be considered for the stable system. These tests should include performance tests on real-life systems.

It was also important to verify that our implementations are correct, so we did not get any erroneous results. The verifications are shown in chapter 5. We conclude by running some tests on the system. The results shows that the LRU/2Q is a better page replacement algorithm than the clock in a multiuser database management system like Derby.

Bibliography

[1] Bernt Johnsen, presentation at JavaZone 2005
http://www3.java.no/JavaZone/2005/presentasjoner/BerntJohnsen/Bernt_Johnsen-DerbyJavaZone.pdf

[2] Derby homepage
http://db.apache.org/derby/

[3] JDBC homepage
http://java.sun.com/products/jdbc/

[4] Opengroup homepage
http://www.opengroup.org/

[5] Andrew S. Tanenbaum,
Modern Operating Systems, second edition, Prentice hall 2001

[6] Hong-Tai Chou, David J. DeWitt
An Evaluation of Buffer Management Strategies for Relational Database System

[7] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum
The LRU-K Page Replacement Algorithm For Database Disk Buffering

[8] Theodore Johnson, Dennis Shasha
2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm

[9] Sorav Bansal and Dharmendra S. Modha
CAR: Clock with Adaptive Replacement

[10] Nimrod Megiddo, Dharmendra Modha
ARC: A Self-Tuning, Low Overhead Replacement Cache

[11] PosgreSQL Directions
http://www.postgresql.jp/misc/seminar/2006-02-17_ 18/materials/01_ Josh_ Berkus.pdf

Appendix A

Code

A.1 Algorithm source code

This section contains the source code for LRU, LRU/2Q as well as the LRUItem

A.1.1 LRU

The source code for LRU.

```
package org.apache.derby.impl.services.cache;

import java.io.FileWriter;
import java.io.IOException;
import java.util.*;

import org.apache.derby.iapi.error.StandardException;
import org.apache.derby.iapi.services.cache.*;
import org.apache.derby.iapi.services.context.ContextManager;
import org.apache.derby.iapi.services.daemon.DaemonService;
import org.apache.derby.iapi.services.daemon.Serviceable;
import org.apache.derby.iapi.services.sanity.SanityManager;
import org.apache.derby.iapi.store.raw.PageKey;
import org.apache.derby.iapi.util.Matchable;
import org.apache.derby.iapi.util.Operator;
import org.apache.derby.impl.store.raw.data.StoredPage;

/**
 * A LRU cache manager. Has one list for the buffer and one
 * hashmap for quick lookup in the list.
 *
 *  Some methods have extra verification methods for debugging
 *  purposes. Derby embedded debugging catches a lot of bugs.
 *  But this is often bundled in a lot of nested exception
 *  handling, and shutdown of the database. Finding the
 *  root (exeception) for these cascading events is not very easy.
 *  The verification methods are designed to let the designer
```

```
 *  know what exactly went wrong.
 *
 * Pages in aIn and aOut are put in holder items called
 * LRUItems. Most important methods are find, release
 * and findfreeItem.
*  Not all interface methods are implemented. As far as we
*  can see many methods never gets called in a page cache
*  manager implementation, like this.
 *
 * @see Servicable
 * @see CacheManager
 * @see LRUItem
 * @see Cacheable
 *
 */



public class LRU extends HashMap
    implements CacheManager, Serviceable
{
    private String name;
    final long MAX_ENTRIES;
    public final CacheStat stat = new CacheStat();
    private LinkedList lru;
    private CacheableFactory holderFactory;
    private boolean active;
    private int usedItems;

    /// Fields only used for debuggin and statistics.
    private int counter=0;
    private int lineNr=0;

    FileWriter f;
    FileWriter lruDump;
    FileWriter findCount;
    int nrFind=0;
    private boolean debug = true;
    private boolean log = true;

    /**
     * Constructor - Currently ignores initial size and goes
     *  right to maximum size.
     *
   *  @param holderFactory the cacheable object class
   *  @param name the name of the cache
   *  @param initialSize the initial number of cachable
   *    object this cache holds.
   *  @param maximumSize the maximum size of the cache.
```

```
 */

public LRU(CacheableFactory holderFactory, String name, int
    initialSize, long maximumSize, boolean useByteCount)
{
  System.out.println("TestLRU");
  System.out.println("initialsize "+initialSize);
  System.out.println("maximumsize "+maximumSize);
    this.name = name;
    MAX_ENTRIES = maximumSize;
    stat.initialSize = initialSize;
    stat.maxSize = maximumSize;
    this.holderFactory = holderFactory;
    initialize(maximumSize);
    active = true;
    usedItems =0;
    try{
      lruDump = new FileWriter("lru.txt");
      lruDump = new FileWriter("lru.txt",true);
    }catch (IOException e) {
  e.printStackTrace();
}
}

/**
 *
 * Initializes the lru list and creates LRUItems.
 *
 * @param initialsize - size of cache
 */
private void initialize(long initialsize)
{
    lru = new LinkedList();
    for(int i = 0; i < initialsize; i++)
    {
        LRUItem item = new LRUItem();
        lru.add(item);
    }

}

////// Interface methods from CacheManager
   ////////////////////////////////////////


/**
 * All items that will be accessed from the
 * cachemanager must first be created by this method.
```

```
     * The page is put in the front of aIn.
     *
     * @see PageKey
     *
     * @param key - Pagekey of page to be created
     * @param createParameter
     */


    public Cacheable create(Object key, Object createParameter)
        throws StandardException
    {
      if(debug)System.out.println("create");

      LRUItem item = (LRUItem)get(key);
      if(item!= null) throw StandardException.newException("XBCA0.
          S", name, key);

        Cacheable entry = null;
        Cacheable oldentry=null;
        findFreeItem(lru,(int)MAX_ENTRIES,this);

        lru.remove(item);
        lru.addFirst(item);

        entry = item.getEntry();
        if(entry != null)
        {
            if(entry.isDirty())
                entry.clean(false);
            if(entry.getIdentity()!=null)
              entry.clearIdentity();

        }

        entry = holderFactory.newCacheable(this);

        oldentry = entry;
        entry = createIdentity(oldentry,key,createParameter);
        item.incKeep();

        item.setEntry(entry);

        put(key, item);
        if(SanityManager.DEBUG)
          return verifyCreate(entry,key,item);
        else return entry;
    }
    /**
```

```
     * Verification method for create, used for debugging
     *
     * @param entry - from create
     * @param key - from create
     * @param item - from cre<ate
     * @return - Cacheable to be returned from create (if no
         errors are found).
     * @throws StandardException
     */
    public Cacheable verifyCreate(Cacheable entry, Object key,
        LRUItem item) throws StandardException{

      int state = 0x000000000000000;

      if(entry==null) state=+ 1;
      if(item.getEntry()!=entry) state=+ 2;
      if(!entry.getIdentity().equals(key)) state=+ 4;
      if(item.keepCount!=1) state=+ 8;
      if(entry.getIdentity() == null) state +=16;
      if(((LRUItem)get(key)).getEntry()!=entry) state+=32;
      if(state!=0){
        System.out.println("Create Errorstate: "+Integer.
            toBinaryString(state));
        throw StandardException.newException("Error in create");
      }

      return entry;

    }

    /**
     * Find an object in cache or fault in from disk.
     * Method contains, together with findfreeItem, the
     * LRU logic. A call to find result in one of two cases.
     *
     * Case1 - Page found in cache. Page is put at front
     * of lru que.
     * Case2 - Page is not found in cache. Page is faulted
     * in and put at the front of lru que.
     *
     *@param key - the key to the object
   *@return a cacheable object that is kept in the cache.
   *@exception StandardException Cloudscape Standard
   *error policy
  */

    public Cacheable find(Object key)
        throws StandardException
    {
```

```
        if(debug)System.out.println("find");
        if(debug)nrFind++;
        Cacheable entry;
        LRUItem item;

        synchronized (this) {

//          Debug code, writing statistics
          counter++;
        if(counter == 10) {
          try {
            f = new FileWriter("lruStat.txt");
            f.write("" + stat+" \n");
            f.close();
            findCount= new FileWriter("findCount.txt");
            findCount.write("#find "+nrFind);
            findCount.close();
          } catch(Exception e){
            System.out.println("Error med skriving til fil");
            e.printStackTrace();
          }
          counter =0;
        }


        // If Derby is in shutdown return null

            if(!active)
                return null;

            item = (LRUItem)get(key);

            if((item!=null)&&(item.getEntry()!=null)){
              entry=item.getEntry();
        }

            if(item == null)
            {
              /// CASE 2 - Item not found in cache
              stat.findMiss++;
              item = findFreeItem(lru,(int)MAX_ENTRIES,this);
                entry=item.getEntry();
                entry =setIdentity(entry,key);
                item.setEntry(entry);
                lru.remove(item);
                lru.addFirst(item);
                item.incKeep();
                put(key, item);
            }
```

```
      else{
        // CASE 1 - Item found
        stat.findHit++;
        entry = item.getEntry();
        lru.remove(item);
        lru.addFirst(item);
        item.incKeep();
      }
      if(SanityManager.DEBUG) return verifyFind(entry,item,key
          );
    else return entry;
  }

}

/**
  * Verification method for find.
 *
 * @param entry - Cacheable to be returned
 * @param item - LRUItem holding entry
 * @param key - Pagekey of entry
 * @return
 * @throws StandardException
 */
public Cacheable verifyFind(Cacheable entry,LRUItem item,
    Object key) throws StandardException{

  System.out.println("used items: "+usedItems);
  try{
    lruDump.write("\n");
    Cacheable testEntry;
    lruDump.write("line:"+lineNr+" ");
    lineNr++;
  synchronized (this) {

    for (Iterator iter = lru.iterator(); iter.hasNext();) {
  LRUItem lruItem = (LRUItem) iter.next();
  testEntry=lruItem.getEntry();

  PageKey pk=null;
  if(testEntry!=null)
    if(testEntry.getIdentity()!=null) pk = (PageKey)testEntry.
        getIdentity();

  long containerId = pk==null ? -1 : pk.getContainerId().
      getContainerId();
  long pageNumber = pk==null ? -1 : pk.getPageNumber();

  lruDump.write("|("+containerId+","+pageNumber+")");
```

```
      }
    }
    PageKey pkEntry = (PageKey)entry.getIdentity();
    lruDump.write("find:("+pkEntry.getContainerId().
        getContainerId()+","+pkEntry.getPageNumber()+")");


    }catch (Exception e) {

      e.printStackTrace();
}
    int state = 0x00000000;

    if(item==null) state+=1;
    if(item.getEntry()!=entry) state+=2;

    if(entry!=item.getEntry()) state+=4;

    if(get(entry.getIdentity())==null) state+=8;
    if(get(entry.getIdentity())!=item) state+=16;

    if(entry.getIdentity()==null) state+=32;

    if(!entry.getIdentity().equals(key)) state+=64;
    if(((LRUItem)get(key)).getEntry()!=entry) state+=128;

    if(item.keepCount<1) state+=256;

    if(lru.size()>MAX_ENTRIES+20) state+=512;

    if(state!=0){
      System.out.println("Error state (find):"+state);//Integer.
          toBinaryString(state));
      throw StandardException.newException("Error in TestLRU, "
          find");
    }

    return entry;
}

/**
 * @return the current maximum size of the cache.
 */

public long getMaximumSize()
{
    return MAX_ENTRIES;
}
```

```
/**
 * Change the maximum size of the cache.
 * @param newSize the new maximum cache size
 */

public void resize(long newSize)
    throws StandardException
{
  System.out.println("resize");

    long currentSize = lru.size();
    if(newSize == currentSize)
        return;
    if(newSize > currentSize)
    {
        for(int i = 0; (long)i < newSize - currentSize; i++){
            LRUItem item = new LRUItem();
          item.setEntry(holderFactory.newCacheable(this));
          lru.addLast(item);
        }
    } else
    if(newSize < currentSize)
    {
        Cacheable entry = null;
        do
        {
          synchronized (this) {

            for(int i=0;i<lru.size();i++){
              LRUItem item = (LRUItem)lru.removeLast();
              entry = item.getEntry();
              if(entry == null)
                  continue;
              if(entry.isDirty())
                  entry.clean(false);
              entry.clearIdentity();
              item =null;
            }
          }try{
            wait();
          }catch(InterruptedException ire){
            throw StandardException.interrupt(ire);
          }
        }while(lru.size()>newSize);
        notifyAll();

    }
}
```

```
/**
 *
 * Find an object already in cache. If not in cache
 * return null.
 * NOTE - Have never seen this method called.
 * @param key - Pagekey of Cacheable we seek.
 * @return Cacheable found.
 */

public Cacheable findCached(Object key)
    throws StandardException
{
  if (debug)System.out.println("findCached");
    LRUItem item = (LRUItem)get(key);
    if(!active)
        return null;
    if(item != null)
    {
        item.incKeep();
        lru.remove(item);
        lru.addFirst(item);
        return item.getEntry();
    } else
    {
        return null;
    }
}

/**
 * Determine whether a key is in the cache.
 *
 *@param key - Pagekey of Cacheable we seek
 *@return true if found.
 */

public boolean containsKey(Object key)
{
  System.out.println("containskey");
    LRUItem item = (LRUItem)get(key);
    return item != null;
}

/**
 *
 * Not implemented, never seen in use
 */

public void setUsed(Object aobj[])
{
```

```
      System.out.println("setUsed");
    }

    /**
  * Release a Cacheable object previously found with find().
  * Releasing a page will decrease its keepCount.
  *@param entry - Page to be released.
*/

    public void release(Cacheable entry)
    {
      if(debug)System.out.println("release");
        LRUItem item = (LRUItem)get(entry.getIdentity());
        int keepCount = item.getKeepCount();

        ///
        synchronized(this)
        {
          item.decKeep();
          if(SanityManager.DEBUG){

            if(debug)verifyRelease(entry,item,keepCount);
          }
        }
    }

    /**
     * Verification method for release.
     *
     * @param entry - Cacheable to be released
     * @param item - LRUItem holding entry
     * @param keep - keepCount before release
     */

    public void verifyRelease(Cacheable entry, LRUItem item,int
        keep){
      boolean ok = true;

    if(keep!=(item.getKeepCount()+1)) ok=false;

      if(item.getEntry()!=null){
        if(entry != item.getEntry()) ok =false;
        if(get(entry.getIdentity())!=null) ok =false;

      }

      if(!ok){
        StandardException.newException("Error in TestLRU, release"
            );
```

```
        }
    }

    /**
     *
     * Explicitly remove a page from cache. Sets
     * RemoveRequested so no one else will try to remove
     * the same item.
     *
     * @param entry - page to be removed
     *
     *  @exception StandardException Standard Cloudscape
     * error policy.
*/
    public void remove(Cacheable entry)
        throws StandardException
    {
      if(debug)System.out.println("remove");

        LRUItem item = (LRUItem)get(entry.getIdentity());
        item.decKeep();
        if(item.isRemoveRequested())
            return;
        synchronized(this)
        {

            item.setRemoveRequested(true);
            do
            {
                if(item.isRemoveRequested())
                    break;
                try
                {
                    wait();
                }
                catch(InterruptedException ire)
                {
                    throw StandardException.interrupt(ire);
                }
            } while(item.getKeepCount() > 0);

            if(entry.isDirty())
                entry.clean(true);

            remove(item.getEntry().getIdentity());
            entry.clearIdentity();

            item.setRemoveRequested(false);
```

```
            }
        }
        /**
     * Cleaning all items the cache. Passes the call to
     * the generic cleanCache method.
     *
     * @see Cacheable#clean
     * @see Cacheable#isDirty
     *
     * @exception StandardException Standard Cloudscape
     * error policy.
    */

        public void cleanAll()
            throws StandardException
        {
          System.out.println("cleanAll");
            stat.cleanAll++;
            cleanCache((Matchable)null);
        }

        /**
     * Clean all objects that match the partialKey (or exact key).
     * Passes the call to the generic cleanCache method.
     *
     *@see Matchable
     @exception StandardException Standard Cloudscape error policy.
    */



        public void clean(Matchable partialKey)
            throws StandardException
        {
          if(debug)System.out.println("clean");
            cleanCache(partialKey);
        }

        /**
     * Age as many objects as possible out of the cache.
     * Method not implemented. This should not have an
     * impact on cache manager performance. As far as we
     * have observed the method is only called during shutdown
     *
     *@see Cacheable#clean
     *@see Cacheable#clearIdentity
    */

        public void ageOut()
```

```
    {

      if(debug)System.out.println("ageOut");
      try {
      f = new FileWriter("Test.txt");
      f.write("" + stat);
      f.close();
    } catch(Exception e){
      System.out.println("Error med skriving til fil");
      e.printStackTrace();
    }
    }

    /**
  Shutdown the cache. This call stops the cache
  returning any more valid references on a find() or
  findCached() call,  and then cleanAll() and ageOut()
   are called. The cache remains in existence until
   the last kept object has been unkept.

  @exception StandardException Standard Cloudscape error policy.

*/


    public void shutdown()
        throws StandardException
    {
      System.out.println("shutDown");
        synchronized(this)
        {
            active = false;
        }
        cleanAll();
        ageOut();
    }

    /**
  *This cache can use this DaemonService if it needs some work to
     be done
  *in the background.
  *Not implemented
*/
    public void useDaemonService(DaemonService daemonservice)
    {
      System.out.println("useDaemonService");
    }
    /**
  *
```

```
  * Throw all items that match the partial key out of cache.
  * @see Matchable
  * @return true if discard has successful gotten rid of
  * all objects that match the partial or exact key.
  * False if some objects that matches were not gotten
  * rid of because it was kept.
*/

    public boolean discard(Matchable partialKey)
    {
      if (debug)System.out.println("discard");
        boolean discardedAll = true;
        synchronized(this)
        {

          for (Iterator iter = lru.iterator(); iter.hasNext();) {

        LRUItem item= (LRUItem) iter.next();
        Cacheable entry = item.getEntry();
        if(entry==null) continue;
        if(item.getKeepCount()==0){
          if(partialKey!=null ){
            if(partialKey.match(entry.getIdentity())){
              remove(entry.getIdentity());
              entry = clearIdentity(entry);

            }
          } else {
            remove(entry.getIdentity());
            entry = clearIdentity(entry);

          }
        }
        else discardedAll = false;
      }
        }
        try {
          FileWriter f = new FileWriter("Test.txt");
      f.write("" + stat);
      f.write("//Cache size: "+MAX_ENTRIES);
      f.close();
    } catch(Exception e){
      System.out.println("Error med skriving til fil");
      e.printStackTrace();
    }
        return discardedAll;
    }

    /**
```

```
 * Report the number of items in use (with Identity)
 * in this cache.
*/

    public int getNumberInUse()
    {
      System.out.println("getNumberInUse");
        int number = 0;
        synchronized(this)
        {
            Iterator iter = lru.iterator();
            do
            {
                if(!iter.hasNext())
                    break;
                LRUItem element = (LRUItem)iter.next();
                if(element.getEntry().getIdentity() != null)
                    number++;
            } while(true);
        }
        return number;
    }

    /**
  Return statistics about cache that may be implemented.
**/


    public long[] getCacheStats()
    {
      System.out.println("getCacheStats");
        return null;
    }


  /**
    reset the cache statistics to 0.
  **/

    public void resetCacheStats()
    {
      System.out.println("resetCacheStats");
    }
    /**
     * Perform an operation on (approximately) all entries
     * that matches the filter, or all entries if the filter
     * is null. Entries that are added while the
     * cache is being scanned might or might not be missed.
     *
```

```
     * @param filter
     * @param operator
     */

    public void scan(Matchable filter, Operator operator)
    {
      System.out.println("scan");
        Cacheable entry = null;
        synchronized(this)
        {
            for(Iterator iter = lru.iterator(); iter.hasNext();)
            {
                LRUItem element = (LRUItem)iter.next();
                entry = element.getEntry();
                Object key = entry.getIdentity();
                if(filter == null || filter.match(key));
            }

        }
        operator.operate(entry);
    }


    //// Methods from Serviceable interface are not implemented.
        ////////////////////////////
    ////  As far as we can see they are not called. May they are
       called if the
    ///  database is left on for a longer period of time.

    public int performWork(ContextManager context)
        throws StandardException
    {
      System.out.println("performWork");
        return 0;
    }

    public boolean serviceASAP()
    {
      System.out.println("serviceASAP");
        return false;
    }

    public boolean serviceImmediately()
    {
      System.out.println("serviceImmediatly");
        return false;
    }
```

```
/// End of interface methods
    ////////////////////////////////////////////////

public void cleanCache(Matchable partialKey)
    throws StandardException
{
     synchronized(this)
    {
        Iterator iter = lru.iterator();
        do
        {
            if(!iter.hasNext())
                break;
            LRUItem element = (LRUItem)iter.next();
            Cacheable entry = element.getEntry();
            if(entry != null&&entry.getIdentity()!=null)
            {
                Object key = entry.getIdentity();
                if(partialKey!=null)
                  if(!partialKey.match(key)) break;
                entry.clean(false);

            }
        } while(true);
    }

}

/**
 * Method containing the logic to remove items from the
 * lru list. The list will have to be traversed to find
 * an LRUItem with keepCount =0. If no items in the list
 * have keepCount = null the list will have to be
 * temporably extended by one (cannot wait for an item
 * to be released as this would break synchronization).
 * Later the list will be shortened to its standard
 * length.
 *
 *
 * @param ll - Linked list we want to find a free
 *  item in.
 * @param maxSize - The standard size of ll. If real
 *  size is more we will try to shorten ll.
 * @param hm - HashMap containing ll's active items.
  * @return LRUItem found
 * @throws StandardException
 */
```

```
public LRUItem findFreeItem(LinkedList ll,int maxSize,HashMap
    hm) throws StandardException
{
  if(debug) System.out.println("findFreeItem");
  if(debug) System.out.println("LL : "+ll.size());
  Cacheable entry = null;
  LRUItem tmp = null;
  LRUItem element = null;

  synchronized (this) {

    /// Traverse the list. Find the last item that can be
        reclaimed.

        for (int f=ll.size()-1;f>=0;f--) {
          tmp = (LRUItem) ll.get(f);
    entry = tmp.getEntry();
    if(tmp.isRemoveRequested()) continue;
    if(tmp.getKeepCount() == 0){

      if(entry==null||entry.getIdentity()==null){
              element=tmp;
              break;
            }

      if(entry.isDirty()) entry.clean(false);

      Object o = hm.remove(entry.getIdentity());


      if(SanityManager.DEBUG){

        SanityManager.ASSERT(o!=null,"Error in findFreeItem,
            no hash-entry");
        if(log){
          try {
            PageKey pk = (PageKey)entry.getIdentity();
            long containerId = pk==null ? -1 : pk.
                getContainerId().getContainerId();
            long pageNumber = pk==null ? -1 : pk.getPageNumber
                ();

            lruDump.write(" evicted:("+containerId+","+
                pageNumber+")");
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
```

```
        element = tmp;
        break;
      }
      element = null;
      tmp = null;

        }

        // Did we find an available item?? If not go to list
           extension.
        if(element!=null){

          //  If list is longer than maxValue (someone extended
          //  the list because there were no unkept items)
          //  we will try to shorten the list.

          // How many items will we try to remove :

          int toLongby = ll.size()-maxSize ;

          tmp = null;
          if(toLongby>0){

            for(int i=ll.size()-1;i>=0;i--){

              if(toLongby==0) break;

              tmp = (LRUItem)ll.get(i);
              if(tmp==element) continue;
              if(tmp.isRemoveRequested()) continue;

              entry=tmp.getEntry();
              if(entry!=null){
                if(entry.getIdentity()!=null){
                  if(tmp.getKeepCount()==0){
                    Object o = hm.remove(entry.getIdentity());
                  if(SanityManager.DEBUG)
                    SanityManager.ASSERT(o!=null,"Error in
                        findFreeItem, no hash-entry");

                    if(entry.isDirty())entry.clean(false);
                    entry = clearIdentity(entry);
                    }
                  else continue;
                }
              }
              ll.remove(tmp);
              tmp = null;
              toLongby--;
```

```
            }
          }
          // return the item
          return element;
        }

        // There are no free items in the list. We have to
            extend the list by one.
        if (debug) System.out.println("No free item found, 
            extending the list");
        element = new LRUItem();
        ll.add(element);
      return element;
      }
}

/**
 * Encapsulating the task of creating identiy for a
 * Cacheable. This method is used if the Cacheable has
 * no previous identity.
 *
 * @param entry - the entry to get an identity
 * @param createParameter
 * @return Cacheable with identity
 */

public Cacheable createIdentity(Cacheable entry,Object key,
    Object createParameter)throws StandardException{

  Cacheable retr = entry.createIdentity(key,createParameter);
  usedItems++;

  return retr;
}

/**
 * Encapsulating the task seting identity for a
 * cacheable. This method is used if the Cacheable has a
 * previous identity, replacing it with a new one.
 *
 * @param entry - Entry to get a new identity
 * @param key - Pagekey that will be the new identity of
 *  entry
 * @return Cacheable with new identity
 * @throws StandardException
 */

public Cacheable setIdentity(Cacheable entry,Object key)
    throws StandardException{
```

```
        System.out.println("setIdentity");
        if(entry==null) {
          entry = holderFactory.newCacheable(this);

        }
        usedItems++;

        Cacheable oldentry;
        oldentry = entry;
        if(entry.getIdentity()!=null){
          if(oldentry.isDirty()) oldentry.clean(false);
            oldentry = clearIdentity(oldentry);

        }
        entry =oldentry.setIdentity(key);
        return entry;
    }

      /**
       * Encapsulating the task of clearing entry of identity
       *
       * @param entry - Cacheable to have identity cleared.
       * @return entry with identity cleared.
       */

      public Cacheable clearIdentity(Cacheable entry) {

        entry.clearIdentity();
        usedItems--;
        return entry;
    }

}
```

A.1.2 LRU/2Q

The source code for LRU/2Q

```
package org.apache.derby.impl.services.cache;

import java.io.FileWriter;
import java.io.IOException;
import java.util.*;

import org.apache.derby.iapi.error.StandardException;
import org.apache.derby.iapi.services.cache.*;
import org.apache.derby.iapi.services.context.ContextManager;
import org.apache.derby.iapi.services.daemon.DaemonService;
import org.apache.derby.iapi.services.daemon.Serviceable;
import org.apache.derby.iapi.services.sanity.SanityManager;
import org.apache.derby.iapi.store.raw.PageKey;
import org.apache.derby.iapi.util.Matchable;
import org.apache.derby.iapi.util.Operator;
import org.apache.derby.impl.store.raw.data.StoredPage;

/**
 * A cache manager based on LRU/2Q. Maintains 3 lists aIn, aM and
 *  aOut. Also has three hashmaps, one for each of the lists.
 *  aIn is a FIFO screening que. Its main job is to hold pages
 *  while corelated references to the page are made. aOut stores
 *  only pointerts to pages that have been in aIn. If a pointer
 *  found in aOut is referenced the page is put into aM, the
 *  main buffer.
 *
 *  Pages in aIn and aOut are put in holder items called
 *  LRUItems. Most important methods are find, release and
 *  findfreeItem.
 *
 *  Some methods have extra verification methods for debugging
 *  purposes. Derby embedded debugging catches a lot of bugs.
 *  But this is often bundled in a lot of nested exception
 *  handling, and shutdown of the database. Finding the
 *  root (exeception) for these cascading events is not very easy.
 *  The verification methods are designed to let the designer
 *  know what exactly went wrong.
 *
 *  Not all interface methods are implemented. As far as we can
 *  see many methods never gets called in a page cache manager
 *  implementation, like this.
 *
 * @see Servicable
 * @see CacheManager
 * @see LRUItem
 * @see Cacheable
```

```
 *
 */

public class LRU2Q
    implements CacheManager, Serviceable
{

    private String name;
    final long MAX_ENTRIES;
    public final CacheStat stat = new CacheStat();
    private LinkedList aM, aIn, aOut;
    private HashMap aMHash, aInHash, aOutHash;
    private int kIn, kOut, kM;

    private CacheableFactory holderFactory;
    private boolean active;
    private int usedItems;



    /// Fields only used for debuggin and statistics.
    private int counter=0;
    private int lineNr=0;

    FileWriter f;
    FileWriter lru2;
    FileWriter findCount;
    int nrFind=0;
    boolean debug=true;
    private boolean log =true;

    /**
     * Constructor - Currently ignores initial size and goes
     * right to maximum size.
     *
   *  @param holderFactory the cacheable object class
   *  @param name the name of the cache
   *  @param initialSize the initial number of cachable
   *  object this cache holds.
   *  @param maximumSize the maximum size of the cache.

     */

    public LRU2Q(CacheableFactory holderFactory, String name, int
        initialSize, long maximumSize, boolean useByteCount)
    {
       this.name = name;
        MAX_ENTRIES = maximumSize;
        stat.initialSize = initialSize;
```

```
        stat.maxSize = maximumSize;
        this.holderFactory = holderFactory;
        kIn= (int)(maximumSize *0.25);
        kOut= (int)(maximumSize* 0.50);
        kM = (int)(maximumSize *0.75);
        aInHash = new HashMap();
        aOutHash = new HashMap();
        aMHash = new HashMap();
        initialize(maximumSize);
        active = true;
        usedItems =0;
        try{
          lru2 = new FileWriter("lru2Q.txt");
          lru2.close();

        }catch (IOException e) {
      e.printStackTrace();
    }
    }

    /**
     *
     * Initializes the lists and creates LRUItems.
     *
     * @param initialsize - size of cache
     */

    private void initialize(long initialsize)
    {
        aM = new LinkedList();
        int aMLength= (int)(initialsize*0.75);
        for(int i = 0; i < aMLength; i++)
        {
            LRUItem item = new LRUItem();
            aM.add(item);
        }

        aIn= new LinkedList();
        for(int i = 0; i < kIn; i++)
        {
            LRUItem item = new LRUItem();
            aIn.add(item);
        }
        aOut= new LinkedList();
    }

    ////// Interface methods from CacheManager ////////////
```

```
/**
 * All items that will be accessed from the cachemanager
 *  must first be created by this method.
 * The page is put in the front of aIn.
 *
 * @see PageKey
 *
 * @param key - Pagekey of page to be created
 * @param createParameter
 */

public  Cacheable create(Object key, Object createParameter)
    throws StandardException
{
  if(debug) System.out.println("create");

  LRUItem aMItem = (LRUItem)aMHash.get(key);
  LRUItem aInItem = (LRUItem)aInHash.get(key);

  if(aMItem!= null || aInItem != null)  throw
      StandardException.newException("XBCA0.S", name, key);

    Cacheable entry = null;
    Cacheable oldentry=null;
    synchronized (this) {

      aInItem = findFreeItem(aIn,kIn,aInHash,aOut,aOutHash,
          kOut);

      aIn.remove(aInItem);
      aIn.addFirst(aInItem);

      entry = aInItem.getEntry();
      if(entry != null)
      {
          if(entry.isDirty())
              entry.clean(false);
          if(entry.getIdentity()!=null)
            entry.clearIdentity();
       }
       entry = holderFactory.newCacheable(this);

       oldentry = entry;
       entry = createIdentity(oldentry,key,createParameter);
       aInItem.incKeep();

       aInItem.setEntry(entry);
      aInHash.put(key, aInItem);
    }
```

```
    if(SanityManager.DEBUG)
      return verifyCreate(entry,key,aInItem);
    else return entry;

}

/**
 * Verification method for create, used for debugging
 * @param entry - from create
 * @param key - from create
 * @param item - from cre<ate
 * @return - Cacheable to be returned from
 *    create (if no errors are found}.
 * @throws StandardException
 */

public Cacheable verifyCreate(Cacheable entry, Object key,
    LRUItem item) throws StandardException{

  int state = 0x000000000000000;

  if(entry==null) state=+ 1;
  if(item.getEntry()!=entry) state=+ 2;
  if(!entry.getIdentity().equals(key)) state=+ 4;
  if(item.keepCount!=1) state=+ 8;
  if(entry.getIdentity() == null) state +=16;
  if(((LRUItem)aInHash.get(key)).getEntry()!=entry) state+=32;
  if(state!=0){
    System.out.println("Create Errorstate: "+Integer.
        toBinaryString(state));
    throw StandardException.newException("Error in create");
  }

  return entry;

}


/**
 * Find an object in cache or fault in from disk. Method
 * contains, together with findfreeItem, the LRU/2Q
 * logic. A call to find result in one of four cases.
 *
 * Case 1 - The object is found in aIn. The page is
 * returned (and kept), but the state of the cache
 * manager remains the same.
 *
 * Case 2 - The Pagekey is found in aOut. The page
```

```
     * is faulted into the front of aM, and the Pagekey
     * removed from aOut.
     *
     * Case 3 - The page is found in aM. Page is moved to
     * front of aM.
     *
     * Case 4 - The page is not found in cache. Page is
     * faulted into the front of aIn. Last item of aIn is
     * removed, and its Pagekey is put in front on aOut.
     * Last Pagekey in aOut is removed.
     *
     *
     *@param key - the key to the object
   *@return a cacheable object that is kept in the cache.
   *@exception StandardException Cloudscape Standard
   *  error policy
  */

    public Cacheable find(Object key)
        throws StandardException
    {
      if(debug) System.out.println("find");
      nrFind++;
      Cacheable entry;
      LRUItem item;
      // If Derby is in shutdown return null
      if(!active)
            return null;

      //Statistikk
      synchronized (this) {
        counter++;


        // Debug code, writing statistics
        if(debug)
        if(counter == 10) {
          try {
            f = new FileWriter("lru2QStat.txt");
            f.write("" + stat+"\n");
            f.close();
            findCount= new FileWriter("findCount.txt");
            findCount.write("#find "+nrFind);
            findCount.close();
          } catch(Exception e){
            System.out.println("Error med skriving til fil");
            e.printStackTrace();
          }
          counter =0;
```

```
    }


    /// CASE 3 Find page in AM.
  /// Move page to front of AM.

    item = (LRUItem)aMHash.get(key);
    if(item != null) {
      //finnes i aM
      aM.remove(item);
      aM.addFirst(item);
      entry = item.getEntry();
      item.incKeep();
      stat.findHit++;
      return verifyFind(item.getEntry(),item,key,aM,aMHash);
    }

    /// CASE 2 Page found in aOut
    /// Page faulted into Am


    PageKey pk = (PageKey)aOutHash.get(key);
    if(pk != null) {
      item = findFreeItem(aM,kM,aMHash,null,null,0);
      entry = item.getEntry();
      if(entry==null)
        entry = holderFactory.newCacheable(this);
      else if(entry.getIdentity()!=null)entry=clearIdentity(
          entry);

      entry = setIdentity(entry,key);
      item.setEntry(entry);
      item.incKeep();

      aM.remove(item);
      aM.addFirst(item);
      aOutHash.remove(key);
      aMHash.put(key,item);
      aOut.remove(key);
      stat.findMiss++;
      return verifyFind(entry,item,key,aM,aMHash);
    }
    // Case 1 - Item found in aIn, do nothing
    item = (LRUItem)aInHash.get(key);
    if(item != null) {
      item.incKeep();
      stat.findHit++;
      return item.getEntry();
    }
```

```
        // CASE 4 - Item not found in cache.
        // Fault in and put in front of Ain.
        item = findFreeItem(aIn,kIn,aInHash,null,null,0);

        entry = item.getEntry();
        PageKey toOut;
        if(entry!=null){
          if(entry.getIdentity()!=null){
            toOut = (PageKey)entry.getIdentity();
            aOut.addFirst(toOut);

            if(aOut.size()>kOut)
              aOutHash.remove((PageKey)aOut.removeLast());
            aOutHash.put(toOut,toOut);

          }
          entry = clearIdentity(entry);
        } else {
          entry = holderFactory.newCacheable(this);
        }

        entry =setIdentity(entry,key);

        aInHash.put(key,item);
        aIn.remove(item);
        aIn.addFirst(item);

        stat.findMiss++;
        item.incKeep();
        item.setEntry(entry);

        if(debug) return verifyFind(entry,item,key,aIn,aInHash);
        return entry;
        }
}

/**
 *
 * Verification method for find.
 *
 * @param entry - Cacheable to be returned
 * @param item - LRUItem holding entry
 * @param key - Pagekey of entry
 * @param ll - LinkedList where page was put
 * @param hm -  HashMap for ll
 * @return returning entry
 * @throws StandardException
 */
```

```
public Cacheable verifyFind(Cacheable entry,LRUItem item,
    Object key,LinkedList ll,HashMap hm) throws
    StandardException{

  if(debug) System.out.println("used␣items:␣"+usedItems);

  if(log){
    try{
      synchronized (this){

        PageKey pk1 = (PageKey) entry.getIdentity();
          if(pk1.getContainerId().getContainerId()==896)
            if(pk1.getPageNumber()==3) entry.toString();


        lru2 = new FileWriter("lru2Q.txt",true);
        lru2.write("\n");
        Cacheable testEntry;
        lru2.write("line:"+lineNr+"␣");
        lineNr++;

        lru2.write("␣aIn:␣");
        for (Iterator iter = aIn.iterator(); iter.hasNext();)
            {
          LRUItem lruItem = (LRUItem) iter.next();
          testEntry=lruItem.getEntry();

          PageKey pk=null;
          if(testEntry!=null)
            if(testEntry.getIdentity()!=null) pk = (PageKey)
                testEntry.getIdentity();

          long containerId = pk==null ? -1 : pk.getContainerId
              ().getContainerId();
          long pageNumber = pk==null ? -1 : pk.getPageNumber()
              ;

          lru2.write("|("+containerId+","+pageNumber+")");

            }

        lru2.write("||␣aOUT:␣");
        for (Iterator iter = aOut.iterator(); iter.hasNext();)
            {
          PageKey pk = (PageKey) iter.next();

          long containerId = pk==null ? -1 : pk.getContainerId
              ().getContainerId();
```

```
            long pageNumber = pk==null ? -1 : pk.getPageNumber()
                ;

            lru2.write("|("+containerId+","+pageNumber+")");


          }
          lru2.write("|| aM: ");

          for (Iterator iter = aM.iterator(); iter.hasNext();) {
        LRUItem lruItem = (LRUItem) iter.next();
        testEntry=lruItem.getEntry();

        PageKey pk=null;
        if(testEntry!=null)
          if(testEntry.getIdentity()!=null) pk = (PageKey)
              testEntry.getIdentity();

        long containerId = pk==null ? -1 : pk.getContainerId().
            getContainerId();
        long pageNumber = pk==null ? -1 : pk.getPageNumber();

        lru2.write("|("+containerId+","+pageNumber+")");

          }

          PageKey pkEntry = (PageKey)entry.getIdentity();
          lru2.write(" |find::("+pkEntry.getContainerId().
              getContainerId()+","+pkEntry.getPageNumber()+")");
        }
        lru2.close();


      }catch (Exception e) {

        e.printStackTrace();
    }
    }

    // Check if number of active items in aIn is the
    // same as in aInHash

    int activeItemsInAin=0;
    for(int i =0;i<aIn.size();i++){

      LRUItem tmpItem = (LRUItem)aIn.get(i);
      Cacheable tmpEntry = tmpItem.getEntry();
      if(tmpEntry!=null)
        if(tmpEntry.getIdentity()!=null)
```

```
            activeItemsInAin++;
        }
        // Check if number of active items in aM is the
        // same as in aMHash

        int activeItemsInAm=0;
        for(int i =0;i<aM.size();i++){

          LRUItem tmpItem = (LRUItem)aM.get(i);
          Cacheable tmpEntry = tmpItem.getEntry();
          if(tmpEntry!=null)
            if(tmpEntry.getIdentity()!=null)
              activeItemsInAm++;
        }


        int state = 0x00000000;

        if(item==null) state+=1;
        if(item.getEntry()!=entry) state+=2;

        if(entry!=item.getEntry()) state+=4;

        if(hm.get(entry.getIdentity())==null) state+=8;
        if(hm.get(entry.getIdentity())!=item) state+=16;

        if(entry.getIdentity()==null) state+=32;

        if(!entry.getIdentity().equals(key)) state+=64;
        if(((LRUItem)hm.get(key)).getEntry()!=entry) state+=128;

        if(item.keepCount<1) state+=256;

        if(debug){
          System.out.println("aInHash: "+aInHash.keySet().size()+"
              aIn: "+activeItemsInAin);
          System.out.println("aMHash: "+aMHash.keySet().size()+" aM:
              "+activeItemsInAm);
          System.out.println("aOutHash: "+aOutHash.keySet().size()+"
              aOut: "+aOut.size());
        }

        if(state!=0){
          SanityManager.ASSERT(state==0,"Error state (find):"+state)
              ;
        }
        return entry;
    }
    /**
```

```
     * @return the current maximum size of the cache.
     */

    public long getMaximumSize()
    {
        return MAX_ENTRIES;
    }

    /**
     * Change the maximum size of the cache.
     * This method is not implemented. As far as we can see
     * this method does is never called for a page
     * cache manager.
     * @param newSize the new maximum cache size
     */

    public void resize(long newSize)
        throws StandardException
    {
      if(debug) System.out.println("resize");

    }

    /**
    * Find an object in the cache. Return null if object not
    * in cache. This method is not implemented. As far as
    * we can see this method does is never called for
    * a page cache manager.
    */

    public Cacheable findCached(Object key)
        throws StandardException
    {
      if(debug) System.out.println("findCached");
      return null;
    }

    /**
     * Determine whether a key is in the cache.
     * This method is not implemented. As far as we can see
     * this method does is never called for a page cache
     * manager.
     */

    public boolean containsKey(Object key)
    {
      if(debug) System.out.println("containskey");
        LRUItem item = (LRUItem)aInHash.get(key);
        LRUItem item2 = (LRUItem)aMHash.get(key);
```

```
        if(item != null)
          return true;
        else if(item2 != null)
          return true;
        else return false;
    }


    /**
     * Mark a set of entries as having been used.
     * This method is not implemented. As far as we can see
     * this method does is never called for a page
     * cache manager.
     *
     * @param keys the key of the used entry.
     */



    public void setUsed(Object aobj[])
    {
      System.out.println("setUsed");
    }

    /**
  * Release a Cacheable object previously found with
  * find(). Releasing a page will decrease
  * its keepCount.
  *@param entry - Page to be released.
*/
    public void release(Cacheable entry)
    {
      if(debug) System.out.println("release");
      synchronized(this)
        {

      LRUItem item = (LRUItem)aInHash.get(entry.getIdentity());
        if( item == null)
          item = (LRUItem)aMHash.get(entry.getIdentity());

        int keepCount = item.getKeepCount();
          item.decKeep();
          notifyAll();
          if(SanityManager.DEBUG){
            SanityManager.ASSERT(!((aInHash.get(entry.getIdentity
                ())!=null)
                &&(aMHash.get(entry.getIdentity())!=null)),"Error,
                    item in both AIn and AOut");
```

```
            verifyRelease(entry,item,keepCount);
          }
        }
    }


    /**
     * Verification method for release.
     *
     * @param entry - Cacheable to be released
     * @param item - LRUItem holding entry
     * @param keep - keepCount before release
     */

    public void verifyRelease(Cacheable entry, LRUItem item,int
        keep){
      boolean ok = true;


      if(keep!=(item.getKeepCount()+1)) ok=false;

      if(item.getEntry()!=null){
        if(entry != item.getEntry()) ok =false;

      }

      if(!ok){
        StandardException.newException("Error in TestLRU, release"
            );
      }
    }


    /**
     *
     * Explicitly remove a page from cache. Sets
     * RemoveRequested so no one else will try to
     * remove the same item.
     *
     * @param entry - page to be removed
     *
     *  @exception StandardException Standard Cloudscape
     * error policy.
*/


    public void remove(Cacheable entry)
        throws StandardException
    {
```

```
    if(debug)
      System.out.println("remove");
    synchronized(this)
      {
      boolean inAIN;
        LRUItem item = (LRUItem)aInHash.get(entry.getIdentity())
            ;
        if(item == null) {
          item = (LRUItem)aMHash.get(entry.getIdentity());
          inAIN = false;
        } else inAIN = true;


        item.decKeep();
        if(item.isRemoveRequested())
            return;
        item.setRemoveRequested(true);

          do
          {
             // Will this work ? Or will waiting break the
                method synchronization
            System.out.println("remove looping");
            try
              {
                  wait();
              }
              catch(InterruptedException ire)
              {
                  throw StandardException.interrupt(ire);
              }

          } while(item.getKeepCount() > 0);

          if(entry.isDirty())
              entry.clean(true);
          if(inAIN)
            aInHash.remove(item.getEntry().getIdentity());
          else aMHash.remove(item.getEntry().getIdentity());
          entry.clearIdentity();
          item.setRemoveRequested(false);

      }
  }


  /**
* Cleaning all items the cache. Passes the call
* to the generic cleanCache method.
```

```
  *
  @see Cacheable#clean
  @see Cacheable#isDirty

  @exception StandardException Standard Cloudscape error policy.
*/


    public void cleanAll()
        throws StandardException
    {
      if(debug) System.out.println("cleanAll");
        stat.cleanAll++;
        cleanCache((Matchable)null);
    }
    /**
  * Clean all objects that match the partialKey.
  * Passes the call to the generic cleanCache method.
  *
  * @see Matchable
  * @exception StandardException Standard Cloudscape
  * error policy.
*/

    public void clean(Matchable partialKey)
        throws StandardException
    {
      if(debug) System.out.println("clean");
        cleanCache(partialKey);
    }


    /**
  * Age as many objects as possible out of the cache.
  * Method not implemented. This should not have an impact
  * on cache manager performance. As far as we
  * have observed the method is only called during shutdown
  *
  *@see Cacheable#clean
  *@see Cacheable#clearIdentity
*/


    public void ageOut()
    {

      if(debug) System.out.println("ageOut");

      if(debug)
```

```
            try {
            f = new FileWriter("Test.txt");
            f.write("" + stat);
            f.close();
          } catch(Exception e){
            System.out.println("Error med skriving til fil");
            e.printStackTrace();
          }

        }


        /**
         *
         * Shutting down the cache. Cache will not return
         * any more pages.
         *
      @exception StandardException Standard Cloudscape error policy.

    */


        public void shutdown()
            throws StandardException
        {
          System.out.println("shutDown");

            active = false;

            cleanAll();
            ageOut();
        }

        /**
         *
         * This cache can use this DaemonService if it needs
         * some work to be done in the background. Method
         * not implemented.
         *

    */
        public void useDaemonService(DaemonService daemonservice)
        {
          System.out.println("useDaemonService");
        }
        /**
      *
      * Throw all items that match the partial key out
      * of cache.
```

```
   * @see Matchable
   * @return true if discard has successful gotten
   *    rid of all objects that match the partial
   *    or exact key. False if some objects that matches
   *    were not gotten rid of because it was kept.
*/


    public boolean discard(Matchable partialKey)
    {
      if(debug) System.out.println("discard");
        boolean discardedAll = true;
        synchronized(this)
        {

          for (Iterator iter = aIn.iterator(); iter.hasNext();) {

        LRUItem item= (LRUItem) iter.next();
        Cacheable entry = item.getEntry();
        if(entry==null) continue;
        if(item.getKeepCount()==0){
          if(partialKey!=null ){
            if(partialKey.match(entry.getIdentity())){
              aInHash.remove(entry.getIdentity());
              entry = clearIdentity(entry);
            }
          } else {
            aInHash.remove(entry.getIdentity());
            entry = clearIdentity(entry);
          }
        }
        else discardedAll = false;
      }

          for (Iterator iter = aM.iterator(); iter.hasNext();) {

        LRUItem item= (LRUItem) iter.next();
        Cacheable entry = item.getEntry();
        if(entry==null) continue;
        if(item.getKeepCount()==0){
          if(partialKey!=null ){
            if(partialKey.match(entry.getIdentity())){
              aMHash.remove(entry.getIdentity());
              entry = clearIdentity(entry);
            }
          } else {
            aMHash.remove(entry.getIdentity());
            entry = clearIdentity(entry);
          }
```

```
        }
        else discardedAll = false;
      }
        }

        return discardedAll;
    }

    public boolean verifyDiscard(boolean discardedAll){

      boolean ok=true;

      return discardedAll;

    }

    /**
 * Report the number of items in use (with Identity) in
 * this cache.
*/

    public int getNumberInUse()
    {
      if(debug) System.out.println("getNumberInUse");
        int number = 0;
        synchronized(this)
        {
            Iterator iter = aIn.iterator();
            do
            {
                if(!iter.hasNext())
                    break;
                LRUItem element = (LRUItem)iter.next();
                if(element.getEntry().getIdentity() != null)
                    number++;
            } while(true);

            iter = aM.iterator();
            do
            {
                if(!iter.hasNext())
                    break;
                LRUItem element = (LRUItem)iter.next();
                if(element.getEntry().getIdentity() != null)
                    number++;
            } while(true);

        }
        return number;
```

```
    }

    /**
  * Return statistics about cache that may
  * be implemented.
**/


    public long[] getCacheStats()
    {
      System.out.println("getCacheStats");
        return null;
    }


  /**
    reset the cache statistics to 0.
  **/

    public void resetCacheStats()
    {
      System.out.println("resetCacheStats");
    }
    /**
     * Perform an operation on (approximately) all entries
     * that matches the filter, or all entries if the filter
     * is null. Entries that are added while the
     * cache is being scanned might or might not be missed.
     *
     * @param filter
     * @param operator
     */

    public void scan(Matchable filter, Operator operator)
    {
      if(debug) System.out.println("scan");
        Cacheable entry = null;
        synchronized(this)
        {
            for(Iterator iter = aIn.iterator(); iter.hasNext();)
            {
                LRUItem element = (LRUItem)iter.next();
                entry = element.getEntry();
                Object key = entry.getIdentity();
                if(filter == null || filter.match(key))
                  operator.operate(entry);
            }

            for(Iterator iter = aM.iterator(); iter.hasNext();)
```

```
            {
                LRUItem element = (LRUItem)iter.next();
                entry = element.getEntry();
                Object key = entry.getIdentity();
                if(filter == null || filter.match(key))
                  operator.operate(entry);
            }

        }

    }

/*
   Methods from Serviceable interface are not implemented.
   As far as we can see they are not called. May they are
   called if the database is left on for a longer
   period of time.
*/
    public int performWork(ContextManager context)
        throws StandardException
    {
      System.out.println("performWork");
        return 0;
    }

    public boolean serviceASAP()
    {
      System.out.println("serviceASAP");
        return false;
    }

    public boolean serviceImmediately()
    {
      System.out.println("serviceImmediatly");
        return false;
    }

    /// End of interface methods //////////////////////////

    /**
     *
     * Method clean the cache for items matching a partial key.
     * If the partial key is null, all items are cleaned.
     *
     *
     * @see Matchable
     */
```

```
public void cleanCache(Matchable partialKey)
    throws StandardException
{
    synchronized(this)
    {
        Iterator iter = aIn.iterator();
        do
        {
            if(!iter.hasNext())
                break;
            LRUItem element = (LRUItem)iter.next();
            Cacheable entry = element.getEntry();
            if(entry != null&&entry.getIdentity()!=null)
            {
                Object key = entry.getIdentity();
                if(partialKey!=null)
                  if(!partialKey.match(key)) continue;
                entry.clean(false);

            }
        } while(true);

        iter = aM.iterator();
        do
        {
            if(!iter.hasNext())
                break;
            LRUItem element = (LRUItem)iter.next();
            Cacheable entry = element.getEntry();
            if(entry != null&&entry.getIdentity()!=null)
            {
                Object key = entry.getIdentity();
                if(partialKey!=null)
                  if(!partialKey.match(key)) continue;
                entry.clean(false);

            }
        } while(true);
    }

}

/**
 * Method containing the logic to remove items from page
 * cache lists. Lists will have to be traversed to find
 * an LRUItem with keepCount =0. If no items in the list
 * have keepCount = null the list will have to be
 * temporably extended by one (cannot wait for an item
```

```
 * to be released as this would break synchronization).
 * Later the list will be shortened to its standard length.
 *
 *
 * @param ll - Linked list we want to find a free
 *  item in.
 * @param maxSize - The standard size of ll. If
 *  real size is more we will try to shorten ll.
 * @param hm - HashMap containing ll's active items.
 * @param overflow - If called from find, Case 1,
 *  this will be aOut. Else null
 * @param OverflowHash - If called from find, Case 1,
 *   this will be aOutHash. Else null
 * @param overFlowSize - If called from find, Case 1,
 *   this will be kOut. Else null
 * @return a free LRUItem
 * @throws StandardException
 */

public LRUItem findFreeItem(LinkedList ll,int maxSize,HashMap
    hm, LinkedList overflow,HashMap OverflowHash,int
    overFlowSize) throws StandardException
{
  if(debug) System.out.println("findFreeItem");
  if(debug) System.out.println("LL : "+ll.size());
  Cacheable entry = null;
  LRUItem tmp = null;
  LRUItem element = null;

  synchronized (this) {



    /// Traverse the list. Find the last item that can be
       reclaimed.

        for (int f=ll.size()-1;f>=0;f--) {
          tmp = (LRUItem) ll.get(f);
    entry = tmp.getEntry();
    if(tmp.isRemoveRequested()) continue;
    if(tmp.getKeepCount() == 0){

      if(entry==null||entry.getIdentity()==null){
              element=tmp;
              break;
            }

      if(entry.isDirty()) entry.clean(false);
```

```
    Object o = hm.remove(entry.getIdentity());

    if(SanityManager.DEBUG){

      SanityManager.ASSERT(o!=null,"Error in findFreeItem,
          no hash-entry");
      if(log){
        try {
          lru2 = new FileWriter("lru2Q.txt",true);
          PageKey pk = (PageKey)entry.getIdentity();
          long containerId = pk==null ? -1 : pk.
              getContainerId().getContainerId();
          long pageNumber = pk==null ? -1 : pk.getPageNumber
              ();

          lru2.write(" evicted:("+containerId+","+pageNumber
              +")");
          lru2.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
    element = tmp;
    break;
  }
  element = null;
  tmp = null;

      }

    // Did we find an available item?? If not go to list
        extension.
    if(element!=null){

      //  If list is longer than maxValue (someone extended
      //  the list because there were no unkept items)
      //  we will try to shorten the list.

      // How many items will we try to remove :

      int toLongby = ll.size()-maxSize ;

      tmp = null;
      if(toLongby>0){

        for(int i=ll.size()-1;i>=0;i--){

          if(toLongby==0) break;
```

```
        tmp = (LRUItem)ll.get(i);
        if(tmp==element) continue;
        if(tmp.isRemoveRequested()) continue;

        entry=tmp.getEntry();
        if(entry!=null){
          if(entry.getIdentity()!=null){
            if(tmp.getKeepCount()==0){
              Object o = hm.remove(entry.getIdentity());

              if(SanityManager.DEBUG)
              SanityManager.ASSERT(o!=null,"Error in
                  findFreeItem, no hash-entry");

            if(entry.isDirty())entry.clean(false);
            entry = clearIdentity(entry);
            }
            else continue;
          }

          // If the list we are working with is aIn, we
              must
          // put the removed element in aOut.
          // If the list is aM, just forget the last item
          if(overflow!=null){
            overflow.addFirst(entry.getIdentity());
            OverflowHash.put(entry.getIdentity(),entry.
                getIdentity());
            if(overflow.size()>overFlowSize){
              Cacheable rmv = (Cacheable)overflow.
                  removeLast();
              if(rmv.getIdentity()!=null)
                OverflowHash.remove(rmv.getIdentity());
            }

          }

        }

        ll.remove(tmp);


        tmp = null;
        toLongby--;
      }
    }
    // return the item
    return element;
```

```
        }

        // There are no free items in the list. We have to
            extend the list by one.
        if (debug) System.out.println("No free item found,
            extending the list");
        element = new LRUItem();
        ll.add(element);
      return element;
      }
  }

  /**
   * Encapsulating the task of creating identiy for a
   * Cacheable. This method is used if the Cacheable has no
   * previous identity.
   *
   * @param entry - the entry to get an identity
   * @param createParameter
   * @return Cacheable with identity
   */

 public Cacheable createIdentity(Cacheable entry,Object key,
     Object createParameter)throws StandardException{

    Cacheable retr = entry.createIdentity(key,createParameter);
    usedItems++;

    return retr;
  }

 /**
  * Encapsulating the task seting identity for a cacheable.
  * This method is used if the Cacheable has a previous
  * identity, replacing it with a new one.
  *
  * @param entry - Entry to get a new identity
  * @param key - Pagekey that will be the new identity
  *   of entry
  * @return Cacheable with new identity
  * @throws StandardException
  */

  public Cacheable setIdentity(Cacheable entry,Object key)
     throws StandardException{

    if(debug) System.out.println("setIdentity");
    if(entry==null) {
      entry = holderFactory.newCacheable(this);
```

```
        }
        usedItems++;

        Cacheable oldentry;
        oldentry = entry;
        if(entry.getIdentity()!=null){
          if(oldentry.isDirty()) oldentry.clean(false);
            oldentry = clearIdentity(oldentry);

        }
        entry =oldentry.setIdentity(key);
        return entry;
    }

      /**
       * Encapsulating the task of clearing entry of identity
       *
       * @param entry - Cacheable to have identity cleared.
       * @return entry with identity cleared.
       */

      public Cacheable clearIdentity(Cacheable entry) {

        entry.clearIdentity();
        usedItems--;
        return entry;
    }

}
```

A.1.3 LRUItem

The source code for LRUItem (holder class used in both LRU and LRU/2Q).

```
package org.apache.derby.impl.services.cache;

import java.io.FileWriter;

import org.apache.derby.iapi.error.StandardException;
import org.apache.derby.iapi.services.cache.Cacheable;
import org.apache.derby.iapi.services.sanity.SanityManager;

/**
 * Holder class for Cacheables, used by the LRU ad LRU/2Q cache
    managers.
 * The class can hold information the Cacheable can not hold
    itself.
 *
 * KeepCount:
 * This is the number of references that are held from outside of
    the
 * cachemanager to a Cacheable. KeepCount is increased by one when
 * someone accesses the Cacheable through find or create. It is
    decreased
 * by one when the Cacheable is released.
 *
 * RemoveRequested:
 * Flagg set when someone tries to remove a Cacheable from the
    cache. If
 * keepCount is above 0, removing will have to wait. The flagg
    stops
 * someone else from trying to remove the same Cacheable.
 */

public class LRUItem {

  // the keepCount
  int keepCount=0;
  // the item we are holding
  Cacheable entry=null;
  // Have someone tried to remove this Cacheable from the cache
  boolean removeRequested=false;
  /**
   * Constructor. Do nothing
   *
   */

  public LRUItem(){
  }
```

```
/**
 * Get the keepCount
 *
 *
 * @return the number of references held to the entry.
 */

public int getKeepCount(){

  if(SanityManager.DEBUG)
    SanityManager.ASSERT(keepCount>=0,"keepCount under 0");


  return keepCount;

}

/**
 * Increase keepCount by one
 */
public void incKeep(){
  keepCount++;
}
/**
 * Decrease keepCount by one
 *
 */
public void decKeep(){
  keepCount--;
}

/**
 * Get this Cacheable entry
 *
 * @return
 */
public Cacheable getEntry() {
  return entry;
}

/**
 * Set a new entry
 * @param entry - new Cacheable
 */
public void setEntry(Cacheable entry) {
  this.entry = entry;
}

/**
```

```
     * Check if someone tried to remove the Cacheable
     * @return
     */
    public boolean isRemoveRequested() {
      return removeRequested;
    }

    /**
     * Set the removeRequested
     * @param remove. true If trying to remove, false when finnished
         removing
     */

    public void setRemoveRequested(boolean remove) {
      this.removeRequested = remove;
    }

}
```

A.2 Test Code

This section contains the source code for setting up and executing the tests used in section 6

A.2.1 CreateTestTables

Source code for setting up the tests.

```
import java.sql.*;

public class CreateTestTables {
  private Connection conn;
  private Statement stmt;


  public CreateTestTables() throws Exception {
    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
    conn = DriverManager.getConnection("jdbc:derby:testdb;create=
        true");
    stmt = conn.createStatement();
    System.out.println("Creating Table1");
    createTable("Table1", 400, 3500);
    System.out.println("Creating Table2");
    createTable("Table2", 4000, 310);
  }

  private void createTable(String tableName, int numberOfItems,
      int textSize)
      throws Exception {
    stmt.execute("Drop Table " + tableName);
    stmt.execute("Create Table " + tableName
        + "( ID Integer NOT NULL PRIMARY KEY, Name VARCHAR(" +
            textSize
        + "))");

    System.out.println("Setter inn personer...");
    String name = RandomStringUtils.randomAlphabetic(textSize);
    for (int i = 0; i < numberOfItems; i++) {
      stmt.execute("INSERT INTO " + tableName + " VALUES(" + i + "
          ,'"
          + name + "')");
    }
  }

  public static void main(String[] args) {
    try {
      new CreateTestTables();
    } catch (Exception e) {
      e.printStackTrace();
```

```
        }
    }
}
```

A.2.2 DerbyTest

Source code for launching the tests.

```
import java.io.*;
import java.sql.*;
import java.util.*;

public class DerbyTest extends Thread {
  private int numberOfPersons;

  private String tableName;

  private int waitSeconds = 60;

  private int numberOfThreads = 20;

  private int numberOfRuns = 200;

  private int[] tArray, eArray;

  private Connection conn;

  private FileWriter f;

  public DerbyTest(String tableName, int numberOfPersons, int test
      ,
       boolean scan, double scanEvery, int pageCacheSize) throws
          Exception {
    this.numberOfPersons = numberOfPersons;
    this.tableName = tableName;
    initializeLists(test);
    initializeFiles(scan, scanEvery, test);

    Properties p = System.getProperties();
    // Setting page size
    p.put("derby.storage.pageSize", "4096");
    p.put("derby.storage.pageCacheSize", "" + pageCacheSize);

    System.out.println("Clock");
    startTest("Clock", test);
    System.out.println("LRU");
    startTest("LRU", test);
    System.out.println("LRU2Q");
    startTest("LRU2Q", test);

    printResults();

  }
```

```
private void startTest(String cacheName, int test) throws
    Exception {
  FileWriter file = new FileWriter("PageCache.txt");
  file.write(cacheName);
  file.close();

  Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
  conn = DriverManager.getConnection("jdbc:derby:testdb;create=
      true");

  for (int i = 0; i < numberOfThreads; i++) {
    DerbyThread d = new DerbyThread(i);
    d.start();
  }
  sleep(waitSeconds * 1000);

  try {
    conn = DriverManager
        .getConnection("jdbc:derby:testdb;shutdown=true");
  } catch (Exception e) {
  }
}

private void initializeLists(int test) {
  if (test == 1) {
    ArrayList<Integer> twenty = new ArrayList<Integer>();
    ArrayList<Integer> eighty = new ArrayList<Integer>();

    int j = 0, k = 0;
    for (int i = 0; i < numberOfPersons; i++) {
      double rand = Math.random();
      if (rand < 0.2)
        twenty.add(j++, i);
      else
        eighty.add(k++, i);
    }
    tArray = new int[twenty.size()];
    j = 0;
    for (Iterator iter = twenty.iterator(); iter.hasNext();) {
      int element = (Integer) iter.next();
      tArray[j++] = element;
    }

    eArray = new int[eighty.size()];
    j = 0;
    for (Iterator iter = eighty.iterator(); iter.hasNext();) {
      int element = (Integer) iter.next();
      eArray[j++] = element;
    }
```

```
  } else if (test == 2) {
    int i = 0;
    tArray = new int[numberOfPersons / 5];
    for (; i < tArray.length; i++) {
      tArray[i] = i;
    }

    eArray = new int[numberOfPersons * 4 / 5];
    for (int index = 0; index < eArray.length; index++) {
      eArray[index] = i++;
    }
  } else if (test == 4) {
    tArray = new int[numberOfPersons];
    for (int i = 0; i < tArray.length; i++) {
      tArray[i] = i;
    }
  }
}

private void initializeFiles(boolean scan, double scanEvery,
    int test) {
  if (test != 4) {
    for (int j = 0; j < numberOfThreads; j++) {
      try {
        f = new FileWriter("Thread" + j + ".txt");

        for (int i = 0; i < numberOfRuns; i++) {
          i++;
          double rand = Math.random();
          if (scan) {
            if (rand < scanEvery) {
              String sql = "Select * FROM " + tableName
                  + "\n";
              f.write(sql);
            }
          }
          if (rand < 0.8) {
            int indeks = (int) (Math.random() * (tArray.length))
                ;
            int verdi = tArray[indeks];
            String sql = "Select * FROM " + tableName
                + " WHERE id=" + verdi + "\n";
            f.write(sql);
          } else {
            int indeks = (int) (Math.random() * (eArray.length))
                ;
            int verdi = eArray[indeks];
            String sql = "Select * FROM " + tableName
                + " WHERE id=" + verdi + "\n";
```

```
            f.write(sql);
          }
        }
        f.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  } else { // test==4
    try {

      for (int j = 0; j < numberOfThreads; j++) {
        f = new FileWriter("Thread" + j + ".txt");

        for (int i = 0; i < numberOfRuns; i++) {
          double rand = Math.random();
          if (scan) {
            if (rand < scanEvery) {
              String sql = "Select * FROM " + tableName
                  + "\n";
              f.write(sql);
            }
          }
          int indeks = (int) (Math.random() * (tArray.length));
          int verdi = tArray[indeks];
          String sql = "Select * FROM " + tableName
              + " WHERE id=" + verdi + "\n";
          f.write(sql);
        }
        f.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }

  }
}

private void printResults() throws Exception {
  FileReader f = new FileReader("clockStat.txt");
  BufferedReader b = new BufferedReader(f);
  System.out.println("Testresultater:");
  System.out.println("Clock:\t" + b.readLine());
  f = new FileReader("lruStat.txt");
  b = new BufferedReader(f);
  System.out.println("LRU\t" + b.readLine());
  f = new FileReader("lru2QStat.txt");
  b = new BufferedReader(f);
  System.out.println("LRU-2Q\t" + b.readLine());
```

```
}

public static void main(String[] args) {
  try {
    System.out.println("Creating test tables...");
    new CreateTestTables();

    System.out.println("Running test 1...");
    for (int j = 1; j <= 10; j++) {
      System.out.println("CacheSize: " + 40 * j);
      for (int i = 1; i <= 10; i++) {
        new DerbyTest("Table1", 400, 1, true, 0.05, 40 * j);
      }
    }

    System.out.println("Running test 2...");
    for (int j = 10; j <= 10; j++) {
      System.out.println("CacheSize: " + 40 * j);
      for (int i = 1; i <= 2; i++) {
        new DerbyTest("Table2", 4000, 1, true, 0.05, 40 * j);
      }
    }

    System.out.println("Running test 3...");
    System.out.println("0%");
    for (int i = 1; i <= 10; i++) {
      new DerbyTest("Table1", 400, 1, false, 0, 80);
    }

    System.out.println("1%");
    for (int i = 1; i <= 10; i++) {
      new DerbyTest("Table1", 400, 1, true, 0.01, 80);
    }

    System.out.println("5%");
    for (int i = 1; i <= 10; i++) {
      new DerbyTest("Table1", 400, 1, true, 0.05, 80);
    }

    System.out.println("10%");
    for (int i = 1; i <= 10; i++) {
      new DerbyTest("Table1", 400, 1, true, 0.1, 80);
    }

    System.out.println("20%");
    for (int i = 1; i <= 10; i++) {
      new DerbyTest("Table1", 400, 1, true, 0.2, 80);
    }
```

```
        System.out.println("Running test 4...");
        System.out.println("0%");
        for (int i = 1; i <= 10; i++) {
          new DerbyTest("Table1", 400, 1, false, 0, 160);
        }

        System.out.println("1%");
        for (int i = 1; i <= 10; i++) {
          new DerbyTest("Table1", 400, 1, true, 0.01, 160);
        }

        System.out.println("5%");
        for (int i = 1; i <= 10; i++) {
          new DerbyTest("Table1", 400, 1, true, 0.05, 160);
        }

        System.out.println("10%");
        for (int i = 1; i <= 10; i++) {
          new DerbyTest("Table1", 400, 1, true, 0.10, 160);
        }

        System.out.println("20%");
        for (int i = 1; i <= 10; i++) {
          new DerbyTest("Table1", 400, 1, true, 0.2, 160);
        }

        System.out.println("Running test 5...");
        for (int j = 10; j <= 10; j++) {
          System.out.println("CacheSize: " + 40 * j);
          for (int i = 1; i <= 7; i++) {
            new DerbyTest("Table1", 400, 4, true, 0.05, 40 * j);
          }
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

A.2.3 DerbyThread

Source code for threads with separate Derby connections.

```
import java.sql.*;
import java.io.*;

public class DerbyThread extends Thread {
  private Connection conn;

  private Statement stmt;

  private ResultSet rs;
  private int threadNumber;

  public DerbyThread(int number) {
    this.threadNumber = number;
    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
      conn = DriverManager.getConnection("jdbc:derby:testdb;create
          =true");
      stmt = conn.createStatement();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void run() {
    try {

      FileReader f = new FileReader("Thread" + threadNumber + ".
          txt");
      BufferedReader b = new BufferedReader(f);
      String txt = b.readLine();
      while (txt != null) {
        boolean s = false;
        if(txt.length()== 20)
          s = true;
        String sql = txt;
        rs = stmt.executeQuery(sql);
        if(s) {
          while(rs.next()) {
            rs.getString(2);
          }
        } else {
          rs.next();
          rs.getString(2);
        }

        txt = b.readLine();
```

```
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```