Caching Patterns and Implementation

Octavian Paul ROTARU

Computer Science and Engineering Department, "Politehnica" University of Bucharest, Romania, Octavian.Rotaru@ACM.org

Abstract

Repetitious access to remote resources, usually data, constitutes a bottleneck for many software systems. Caching is a technique that can drastically improve the performance of any database application, by avoiding multiple read operations for the same data.

This paper addresses the caching problems from a pattern perspective. Both Caching and caching strategies, like primed and on demand, are presented as patterns and a pattern-based flexible caching implementation is proposed.

The Caching pattern provides method of expensive resources reacquisition circumvention. Primed Cache pattern is applied in situations in which the set of required resources, or at least a part of it, can be predicted, while Demand Cache pattern is applied whenever the resources set required cannot be predicted or is unfeasible to be buffered.

The advantages and disadvantages of all the caching patterns presented are also discussed, and the lessons learned are applied in the implementation of the pattern-based flexible caching solution proposed.

Keywords

Caching, Caching Patterns, Data Access, Demand Cache, Performance Optimization, Primed Cache, Resource Buffering.

Caching Fundaments

A significant portion of an application's resources is consumed by I/O operations, which usually are data accesses. The data access operations are usually the bottlenecks of any software system, especially if they require network transport as well. It is therefore required to implement the data access modules and/or components as efficient as possible.

Caching is a technique that can drastically improve the performance of any database application. Due to caching multiple read operations for the same data are avoided. Usually, the I/O operations are the most expensive operations that an application can perform, and therefore it is beneficial to limit their use as much as possible.

Waiting indefinitely for remote resources to respond blocks the application during that time. Separating the database access in a separate thread is a good programming practice; even it involves extra-costs of thread management and synchronization.

Minimizing as much as possible the interactions of the application with the exterior will improve its performance. However, there is always a price that must be paid for the performance gain obtained by minimizing the interactions with the exterior.

Caching the data that the application is accessing will increase the memory usage, sometimes beyond acceptable limits. Therefore it is very important to obtain a proper balance between the I/O accesses and the memory usage. In case the memory of the machine is not big enough to host the cached data, the system will start to use the swap and to move pages to the virtual memory (on the hard-disk drive). The benefits of caching will fade away, and in some circumstances the performance of the system can even deteriorate beyond the point of not using caching at all.

The quantity of data being cached and the moment when to load, either in the beginning when the application initializes or whenever it is required for the first time, depending on the requirements of each application.

If the data requirements of and application can be anticipated at the beginning of the run, then loading the data identified as required from the beginning is the best strategy. However, if the data requirements are unpredictable, the only way to function is to load the data first time when required. Refining a caching mechanism to the extent of properly deciding if to add the retrieved data to the cache or not, or which data to purge from the cache whenever the cache is reaching its size limit, requires to study the characteristics of the data in

order to be able to categorize its importance. This kind of tuning is usually required only in time critical applications in which the classical approach of purging the least used data or the oldest data is not enough. Caching is more or less similar with the way an operating system manages memory, especially in case the memory requirements exceed the physical memory installed on that machine. The pages, which are accessed more frequently, are stored in the physical memory and those, which are infrequently accessed, are sent to the virtual memory, a hard disk extension of the main memory.

The main purpose of a database application designer is to achieve an optimal balance between performance and resources. A well-designed database access framework will allow a seamless switch between the cached data and the on-demand data, and different caching configurations and technique.

Caching as Pattern

Patterns are experience-capturing methods, able to share it in a consistent manner with others. They are solutions based on experience to recurrent problems, describing best practices and proven designs.

Patterns originate on Christopher Alexander's work on architectural design [1]. Christopher Alexander considers that "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice" [1]. Caching can boost the performance of an application. As a general performance improvement technique caching is not limited to data, but to any kind of resources that a software system can repeatedly require. As a repetitive performance improvement solution for software system that accesses the same resources multiple times, caching can be formalized as a pattern, as described bellow.

Context

A software system is accessing multiple times the same resources, usually database information. The performance of the system is not satisfactory, and therefore it requires optimization.

Problem

The repetitive acquisition and release of the same resource introduces overhead affects the overall performance of the system. Usually, the resource acquired is information from a remote data source. The overhead of multiple acquisitions affects both CPU and I/O operations.

Memory management is also affected by repetitious initialization and release of the same resource. Dynamic memory allocation is usually a heavy operation, involving many CPU cycles.

The system performance can be improved by reducing the resource management cost (CPU, I/O and memory management).

Forces

The following forces needs to be resolved:

- *Performance* is the main force that needs to be resolved in the above-described context. If the performance constraints of an application are very tight, releasing and acquiring the same resource several times will hinder its performance, and therefore must be avoided.
- *Complexity* - The solution should be kept as simple as possible. Introducing extra complexity can make the solution cumbersome and even affect performance.
- *Memory Usage* - Memory usage is also an important force that needs to be taken into account. No system has unlimited memory and therefore memory consumption should be handled with care.
- *Validity* refers to the lifetime of an acquired resource. Some resources will become obsolete after some time, especially if we are speaking about data resources that can be modified by multiple clients. If some resources, for example file handles, can be hold as long as required, others are time sensitive, like data for examples.

Solution

The solution to the above stated problem is to store the already acquired resources in a buffer, which will allow their reuse. Whenever requested again, the resources will be taken from the cache, without re-acquiring them. The cache will identify the resources using unique identifiers.

When the resources stored in the cache are no longer required they could be released in order to lower the memory usage. Cached resources can be released any time, according to the caching implementation and strategy used. Since the memory is limited, the size of the cache is limited as well, fitting only a limited number of resources.

Deciding which resource to remove from cache whenever its maximum size is reached is a very important decision, which will influence the performance of the system. It is therefore required to make a proper choice in this are. The caching decision support can become kind of complicated, collecting usage statistics or any other information that can be used as decision support.

The Caching pattern allows faster access to resources or information that is frequently accessed, resulting in improved overall performance and scalability of the system. However, this is usually achieved by introducing extra complexity.

In case the cached data is modified by the system and synchronization is required with the external data sources, it is possible that data changes are lost in case of system crash. [2], by Kircher and Jain, for an in-depth presentation and analysis of the Caching pattern can be consulted.

Caching Patterns

Basically, there are two main caching strategies that can be described as patterns: Primed Cache and Demand Cache [3].Choosing the proper strategy for your application will highly affect its performance. Apart from the two caching patterns mentioned earlier, Primed Cache and Demand Cache, [3] presents other patterns as well that provide predefined solutions to different caching related problems: Cache Accessor, Cache Search Sequence, Cache Collector, Cache Replicator, and Cache Statistics.

Ideally, all the data required to perform a use-case implemented by a software system should be retrieved in one access and for most of the software systems the only way to achieve something like this is by caching.

If the data required performing a certain use-case is known from the beginning, than the system can definitely cache it before the run starts. A cache that is initialized from the beginning with default values is called Primed Cache.

However, in case the data required by a use-case run can vary and cannot be cached, what the system can do improve the performance is to bring the data into the memory whenever required and to keep it for future use. Practically, the system relies on the supposition that the same use-case or another one having similar data requirement will soon be executed, and therefore the cache data can become useful. This situation corresponds to the Demand Cache pattern.

Demand Cache loads information from the database and stores in cache whenever the respective information is requested by applications („lazy load"). Therefore, the probability to find the required data in cache grows with each data load, due to cache data accumulation. A Demand Cache implementation will improve performance while running.

Any cache mechanism functions correctly as long as the data that is its subject changes very rare. In fact, the cache mechanism will have maximum performance the data is unchanged during the application run. Any change that is committed in the database and affects the same set of data that was cached will invalidate the cached data, unless the application is not inside a transaction with a high isolation degree, and therefore the changes done after its beginning are not visible to it.

Figure 1 presents the activity flow of caching mechanisms. Any request of data from cache requires the verification of the information validity, unless the data was retrieved during the same transaction. In case data become obsolete, a reload from the database is required. Due to memory limitations it is required to implement a decision mechanism, which will manage the data introduction to and removal from cache.

Primed Cache

A Primed Cache should be considered whenever it is possible to predict a subset or the entire set of data that the client will request, and to prime it to the cache. Figure 2 presents the class diagram of a Primed Cache. The <PrimedCacheAccessor> will maintain a list of partial keys that is primed to avoid repeated priming operations of the same partial keys. Primed Cache has minimal access overheads, having a large quantity of data in the cache. Also, if the data requirements of the application are correctly guessed, then the cache will also occupy an optimum quantity of memory, containing only relevant data.

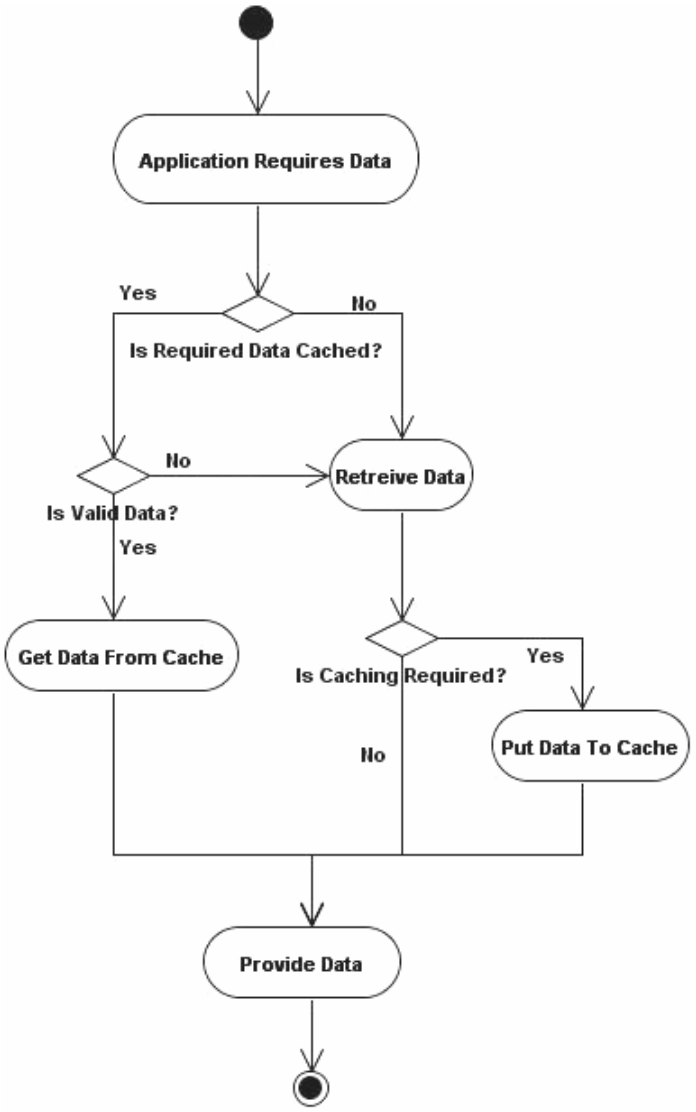

Figure 1. Caching Activity Flow Diagram

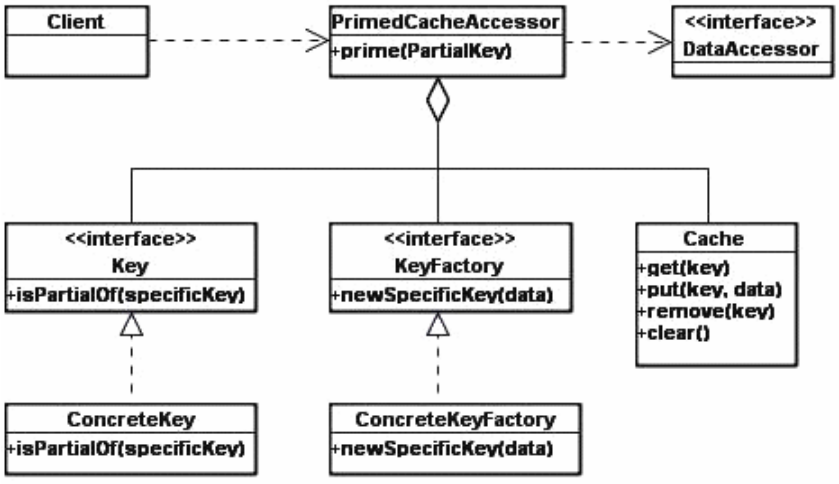

Figure 2. Primed Cache – Class Diagram

In case the client request data with a key that matches one of the primed keys having no corresponding data in the cache it is assumed that there is no data in the data source for that key as well.

If a cache client makes a request based on a key that matches one of the primed keys but has no data in the cache, then for that key there is no data in the database as well.

Whenever the cache receives a data request, it verifies based on the key if the data already exist (Figure 3, operation 2). If the verification result is negative, data is retrieved from the database (Figure 3, operation 4). A new key is created for data retrieved from the database (Figure 3, operations 6, 7). Data is added to the cache (Figure 3, operation 9) and its key is added to the list of cache keys (Figure 3, operation 11).

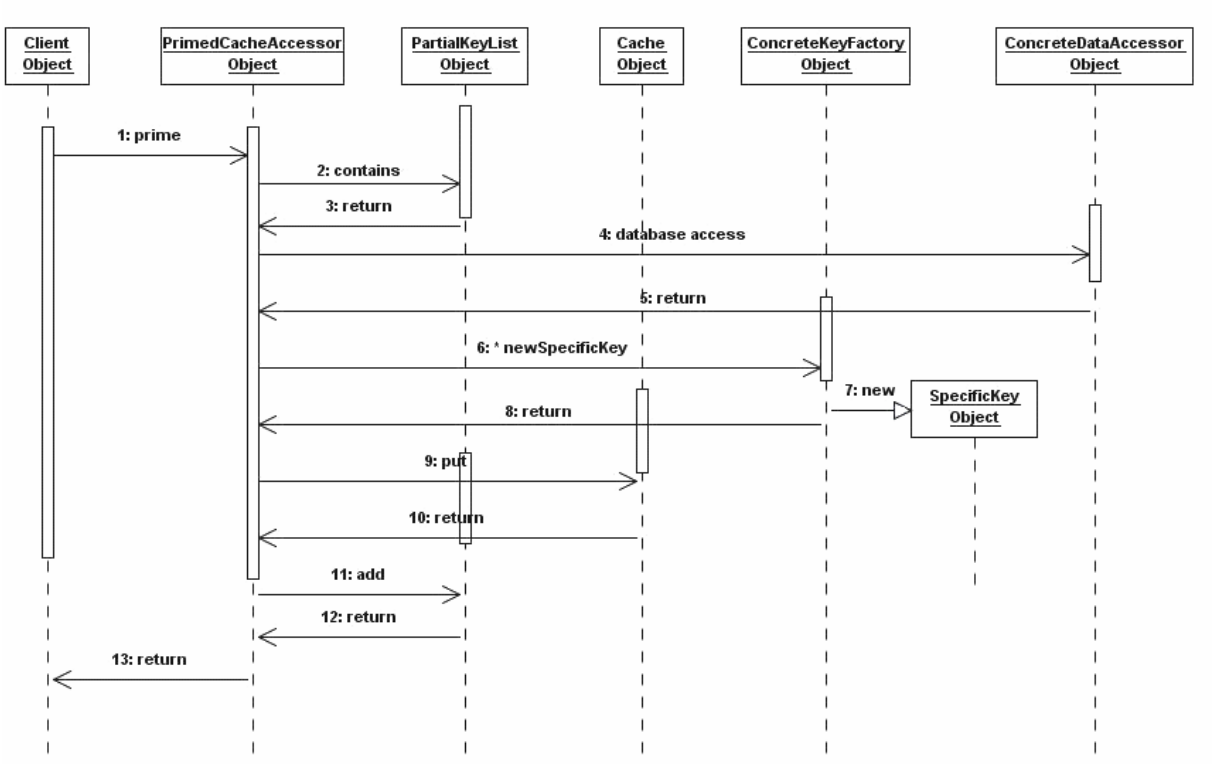

Figure 3. Primed Cache Initialization using partial keys

Demand Cache

A Demand Cache should be considered a solution whenever populating the cache with the complete data set required, or even with a part of it, is either unfeasible or unnecessary.

Figure 4 describes the class structure of a Demand Cache. <CacheAccessor> is the entry point of all the data access operations, managing both data access operations and caching.

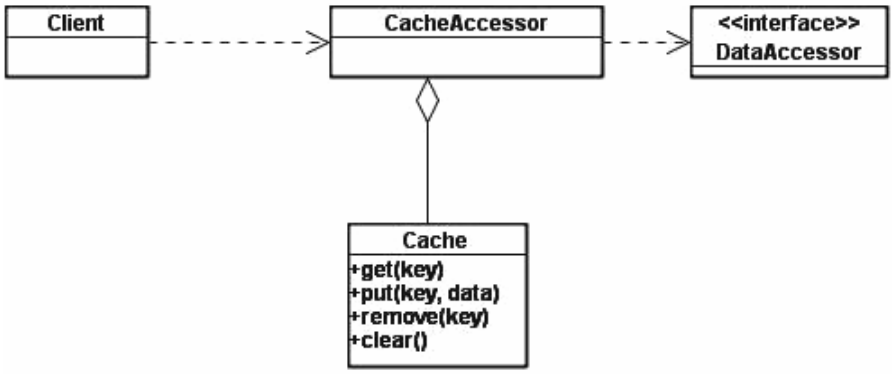

Figure 4. Demand Cache – Class Diagram

When a client requests data, a Demand Cache will function as pictured in Figure 5. The <CacheAccessor> will interrogate the cache about the required data (Figure 5, operation 2). If there is a hit and the data is already cached, the <CacheAccessor> will provide the data returned by the cache to the Client (Figure 5, Cache Hit, operations 3, and 4).

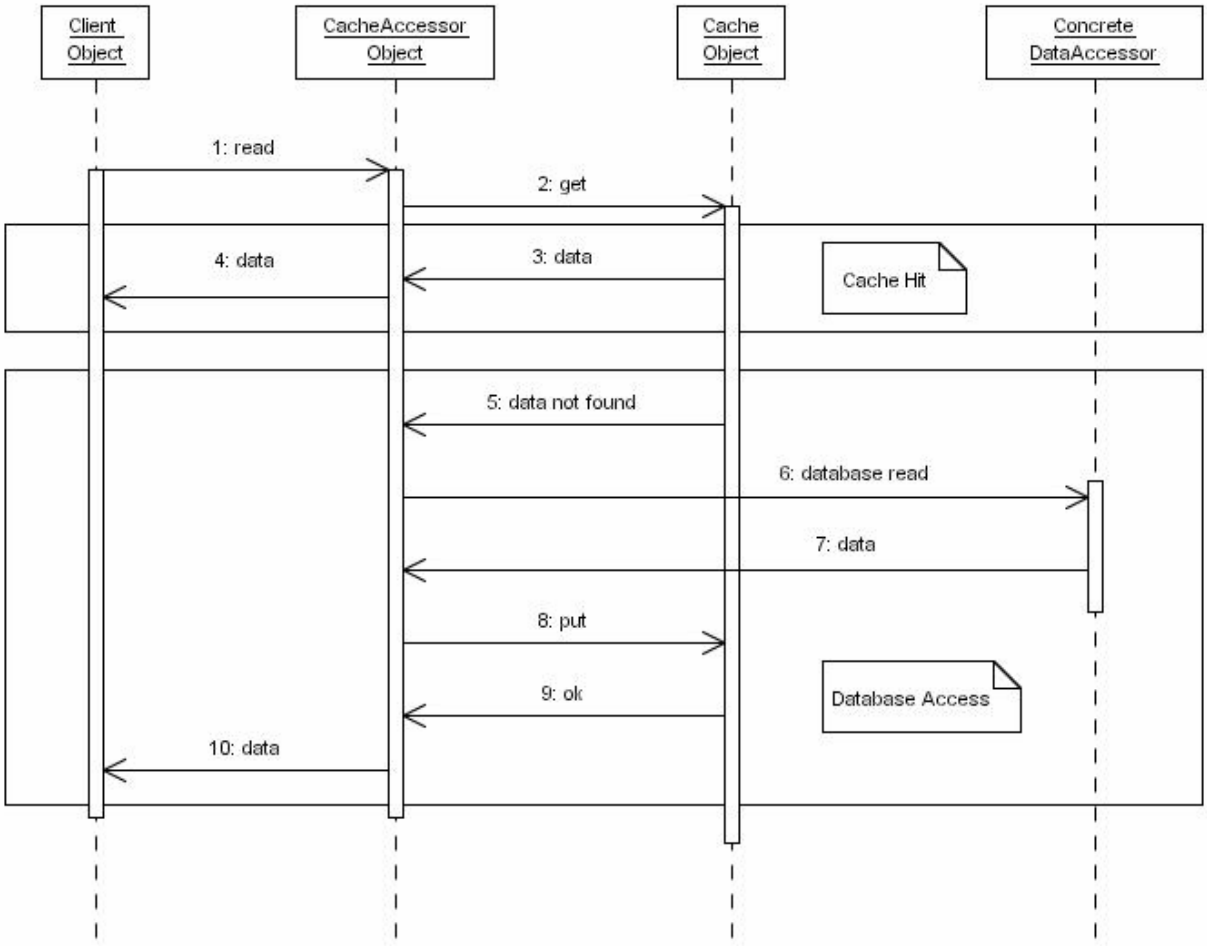

Figure 5. Demand Cache – Sequence Diagram

In case the data is not found in the database, the <DataAccessor> will retrieve it from the database (Figure 5, operation 6), store in the cache as well for future use (Figure 5, operation 8), and present it to the customer.

Even if the application will initialize very fast, a Demand Cache will be populated slow, using many data access operations. The performance of the application is improving during its execution; the probability to have a cache hit growing after each data access operation.

Even if the initialization of the application will be very fast compared with the situation in which a Primed Cache is used, a Demand Cache is getting populated slowly, using many data access operations. The performance of an application that uses a Demand Cache is getting better during the execution; the cache hit probability growing after each data access.

A Pattern Based Flexible Caching Solution

In most of the industrial applications there are use-cases for which the required data is known from the beginning, use-cases for which data cannot be anticipated and in between situations in which a subset of the data can be anticipated a primed.

The above-described situation requires a difficult decision. Implementing a Demand Cache will disadvantage the use-cases for the data set, or at least a part of it, can be anticipated. Also, a Primed Cache will be totally un-useful for use-cases with unpredictable data requirements.

A generic solution that will serve the requirements of most of the applications needs to combine the benefits of both Primed Cache and Demand Cache, with a minimal overhead.

The proposed caching solution presented in Figure 6, uses two different classes to tackle with the two caching situations: <PrimedDataModel> corresponding to Primed Cache pattern and <OnDemandDataModel> corresponding to Demand Cache pattern.

A separate class is used for each table or persistent class that is accessed by the application. In this way, if the database is relational, the database relations are translated into classes, insuring the translation between the two paradigms.

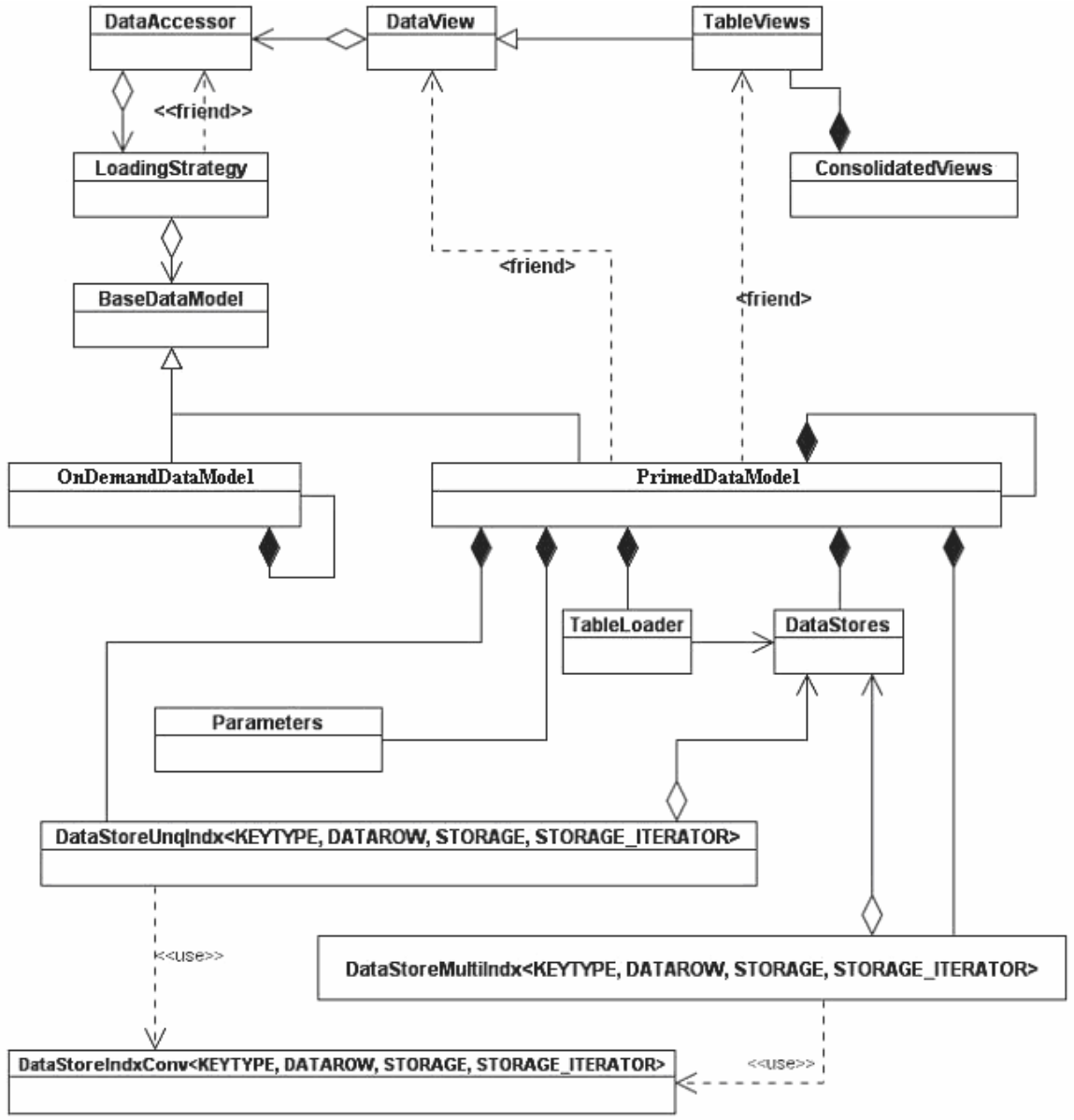

Figure 6. The Proposed Flexible Caching Solution – Class Diagram

All the table classes have the same ancestor: the <DataView> class. Also, the table classes can be combined into consolidated data views, comprising data coming from multiple tables.

The <DataView> class is accessing data using the <DataAccessor> class as a proxy. The <DataAccessor> is the only data provider for the table classes. The <DataAccessor> is loading the data from the cache or from the database on demand using a <LoadingStrategy> class. The <LoadingStrategy> class provides the switch between the cached data and the on-demand data access, by choosing the correct <DataModel>.

The loading strategy is the link between the <DataAccessor> and the <DataModel>. The <BaseDataModel> class is derived into two <DataModel> classes: a cached one, <PrimedDataModel> class, and a on demand one, <OnDemandDataModel>.

Both <PrimedDataModel> and <OnDemandDataModel> are derived from <BaseDataModel> and are singletons, being initialized at the first call. Apart from the object initialization, the data initialization of <PrimedDataModel> is done upon request, during the initialization sequence of its clients.

All data members and methods of <PrimedDataModel> are protected. The data views access their corresponding data containers from <PrimedDataModel> using friend permission.

The <CachedDataModel> contains <DataStores>, which are data containers, and indexes. The <DataStores> are loaded using the <TableLoader> class. The <Parameters> class contains the cache parameters (size, caching decision, etc…).

Indexes are used to facilitate search by keys or sub-keys into the cached data. The decision to index or not a certain data set, or what will be the index criteria is primed, and therefore it needs to be anticipated. Instead of scanning the entire container every time a data request is made, an index can be used for container direct access based on binary search.

STL structures (map and multimap) are used by both <DataStoreUnqIndx> and <DataStoreMultiIndx> to implement the binary search mechanism. <DataStoreUnqIndx> is a unique index structure, each key value having only one corresponding record. <DataStoreUnqIndx> internally uses STL map, which is in turn organized as a binary tree.

The <DataStoreUnqIndx> class is a template class and has the following definition:

```
template <class KEYTYPE, class DATAROW, class STORAGE, class STORAGE_ITERATOR>

class DataStoreUnqIndx {
    STORAGE *                  m_pDS;
    map<KEYTYPE, DATAROW * >   m_mapDSIndx;
public:
    DataStoreUnqIndx( ) { m_pDS = NULL;}

    DataStoreUnqIndx( STORAGE * ds, const TCHAR * pszKeyColumn )
    {      Prepare ( ds, pszKeyColumn );     }

    // prepare index
    void Prepare( STORAGE * ds, const TCHAR * pszKeyColumn,
              TDsIndxConv <KEYTYPE, DATAROW, STORAGE, STORAGE_ITERATOR> * pConv )
    {      ASSERT(ds);   m_pDS = ds;   m_mapDSIndx.clear();
           pConv->SetKeyCol(pszKeyColumn);
           for (STORAGE_ITERATOR it = pConv->Begin(ds);
                it! = pConv->End(ds);
                ++it)
                  m_mapDSIndx[pConv->GetKeyValue(it)] = pConv->GetDataRow(it);
    }
```

```
    // get data row by key value
    DATAROW * operator[] (KEYTYPE key)
    { if (m_mapDSIndx.find(key) != m_mapDSIndx.end())
          return m_mapDSIndx[key];
      else
          return NULL;
    }
};
```

<DataStoreUnqIndx> is instantiated based on 4 parameters: index key type (KEYTYPE), record type (DATAROW), container type (STORAGE), and container iterator to be used (STORAGE_ITERATOR). The iterator type is not assumed to be DATASTORE::iterator in order to give the possibility to chose a different iterator, like for example a constant iterator or a reverse one.

The map data structure is initialized during the construction of the index by calling <Prepare>. The indexed records can be easily referenced using the subscript operator („[]”).

<DataStoreMultiIndx> is a multi index class that allows multiple records for the same value of the key. <DataStoreMultiIndx> uses STL multimap and provides the required mechanisms for the iteration of the records having the same key. <DataStoreMultiIndx> class has the following definition:

```
template <class KEYTYPE, class DATAROW, class STORAGE, class STORAGE_ITERATOR>

class DataStoreMultiIndx {
    STORAGE *     m_pDS;
    multimap<KEYTYPE, DATAROW * >     m_mmapDSIndx;
public:
    DataStoreMultiIndx( ) { m_pDS = NULL;}
    DataStoreMultiIndx( STORAGE * ds, const TCHAR * pszKeyColumn) {
          Prepare ( ds, pszKeyColumn); }

    // prepare index
void      Prepare(     TDatastore * ds,
const TCHAR * pszKeyColumn,
TDsIndxConv <KEYTYPE, DATAROW, STORAGE, STORAGE_ITERATOR> * pConv )
    {
          ASSERT(ds);
          m_pDS = ds;
          m_mmapDSIndx.clear();
          pConv->SetKeyCol(pszKeyColumn);
          for (  STORAGE_ITERATOR it = pConv->Begin(ds);
                it != pConv->End(ds);
                ++it)
                m_mmapDSIndx.insert(pair< KEYTYPE, DATAROW *>(
                      pConv->GetKeyValue(it),
                      pConv->GetDataRow(it)
                );
    }

unsigned  CountByKey(KEYTYPE key)
{
          return m_mmapDSIndx.count(key);
}
```

```
    // key iterator – iterators on values having the same key
    typedef typename multimap<KEYTYPE, DATAROW *>::iterator key_iterator;
    // returns the first element having a certain key
    key_iterator  Begin(KEYTYPE key)
{
        return m_mmapDSIndx.lower_bound(key);
}
    key_iterator  End(KEYTYPE key)
{
        return m_mmapDSIndx.upper_bound(key);
}
};
```

<DataStoreMultiIndx> has the same initialization parameters like <DataStoreUnqIndx>. The records that correspond to the same key can be iterated using <key_iterator>, actually a type definition based on the multimap iterator, the iteration limits being <Begin> and <End> methods that receive that key as parameter.

Passing the storage as an initialization parameter to the indexes insures the genericity of the solution. Data container manipulation routines are also outside the index class, therefore the function that initializes the index (<Prepare>) receives as argument a converter, in fact an abstract interface that needs to be implemented by the clients using it and that contains all the operations that cab be container specific. The converter class has the following definition:

```
template <class KEYTYPE, class DATAROW, class STORAGE, class STORAGE_ITERATOR>

struct DataStoreIndxConv
{
    virtual STORAGE_ITERATOR   Begin(STORAGE & s) = 0;
    virtual STORAGE_ITERATOR   End(STORAGE & s) = 0;
    virtual KEYTYPE            GetKeyValue( STORAGE_ITERATOR it,
const TCHAR * pszKeyColumn = NULL) = 0;
    virtual DATAROW *          GetDataRow(STORAGE_ITERATOR it) = 0;
    void                       SetKeyCol( const TCHAR * pszKeyColumn)
    {
        ASSERT(pszKeyColumn);
        m_pszKeyColumn = pszKeyColumn;
    }
private:
    const TCHAR * m_pszKeyColumn;
};
```

The implementation of <IndexConverter> class, which acts as a proxy for the data container, requires only the encapsulation of the methods offered by the container in the container's standard interface. This is the reason for which the indexing mechanism can use any kind of container as long as <IndexConverter> class is specialized for each type of container used.

Conclusions

This paper demonstrates that the advantages of both Primed Cache and Demand Cache patterns can be combined in a flexible caching implementation. The starting point was the assumption that for most of the applications that repeatedly acquire resources the context do not corresponds neither to Primed Cache nor Demand Cache. It is highly unlikely that the entire set of resources, or at least majority of it, that an application will require can be predicted. In practice, a limited part of the required resources are predictable while all the other, even if accessed repeatedly, will require on demand load.

The proposed flexible solution gathers both Primed Cache and Demand Cache under the same umbrella, trying to combine their advantages. As a possible future work derived from here, I believe that the mixed caching solution presented here can also be formalized as a pattern.

References

[1] Alexander C., Ishikawa S., Silverstein M., *A Pattern Language*, Oxford University Press, 1977.

[2] Kircher M., Prashant Jain, *Caching*, EuroPLOP, 2003.

[3] Nock C., *Data Access Patterns – Database Interactions in Object-Oriented Applications*, Addison-Wesley, Pearson Education, 2004.

[4] Alexander C., *The Timeless Way of Building*, Oxford University Press, 1979.

[5] Kircher M., Jain P., *Pattern-Oriented Software Architecture, Patterns for Resource Management*, John Wiley & Sons, 2004.

[6] Buschmann F., Meunier R., Robnert H., Sommerland P., Stal M., *Patterns-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996.

[7] Fowler M., Rice D., Foemmel M., Hieatt E., Mee R., Statford R., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002.

[8] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[9] Noble J., Weir C., *Small Memory Software: Patterns for Systems with Limited Memory*, Addison-Wesley, 2001.

[10] Martin R. C., Riehle D., Buschmann F. (eds.), *Pattern Language of Program Design 3*, Addison-Wesley, 1998.