Basis Modeling

Technical
Documentation

LiveCache

Basis Modeling Team

For internal use only

SAP AG
Neurottstraße 16
D-69190 Walldorf
Germany

50030817

May 1999

BM046.01

Table Of Contents

Table Of Figures

The following figures can be found at the end of the document:

Preface

This Bluebook provides an overview of the liveCache version 7.2, focusing on the object management system (OMS) in particular[1]. The first release of the liveCache was version 7.1. The Bluebook is intended for readers who want to learn about the concepts as well as internal structure of the liveCache.

Chapter 1

The first chapter gives an overview of the basic concepts of the liveCache. It discusses the interaction with the R/3 System, the rough structure of the liveCache, the general concepts like the OMS application programming interface (API), the consistent read and the OMS transaction concepts.

Chapter 2 and Chapter 3

The second and third chapters examine the liveCache architecture in more detail. The architecture of the liveCache can be divided into two layers. These are the client communication layer, which implements the top-level components of the liveCache, and the liveCache Basis, which implements the liveCache kernel functions and structures. Chapter 2 describes the client communication layer with its SQL and OMS components, chapter 4 the liveCache basis layer with the transaction manager, the data access agent as well as the log access agent and also the implementation of the consistent read.

Chapter 4

The fourth chapter finally deals with the process and thread architecture of the liveCache, discussing the liveCache- and the XServer process. The liveCache process contains many different threads which are discussed including their tasks. Furthermore an attempt is made to map the different threads and their tasks to the components in the software architecture of the liveCache.

Concepts

This report describes the liveCache from a more technical view. However, there are several parts which are recommended for readers who are only interested in the concepts of the liveCache. Chapter 1 completely describes the more conceptual behavior of the liveCache. In chapter 2, sections 2.3, 2.4 and 2.5 might be interesting for a conceptual overview. Section 2.3 describes the OMS request manager with the different operations on persistent objects and the version concept. In 2.4 the handling of commit and rollback on transactions is described and section 2.5 introduces the concepts of how and when an application context is instantiated. In chapter 3, mainly section 3.2, which describes the transaction manager, is recommended. Here the consistent read (refer to 3.2.1) and the lock management (refer to 3.2.2) are described in more detail. In section 3.1 the components of the liveCache basis are roughly described. This might also be interesting for readers which are interested in a more conceptual overview. The later sections describe these components in detail. Chapter 4 only describes the processes and

[1] At the time of writing, the liveCache version 7.2 is still under construction, so changes may still occur.

threads within the liveCache and is therefore not intended for readers who are only interested in the concepts.

Basis Modeling Group

The publications of the Basis Modeling Group as well as further material about the R/3 Basis System and related topics can be found at the following URL: http://ency.wdf.sap-ag.de:1080/. A list of already published Bluebooks is available at http://ency.wdf.sap-ag.de:1080/bluebooks.htm.

1. Introduction

This chapter examines the basic concepts of the liveCache. An overview of the goals of the liveCache is followed by a description of how the liveCache operates with the R/3 System. Furthermore, the rough structure of the liveCache and the transaction concepts are explained.

What is the liveCache ?

The liveCache is a result of the ADABAS D Database Management System Version 6.2 extended with the capability to handle persistent objects provided by the so-called object management system (OMS). The first goal of the liveCache is to handle all data in main memory, so that retrieval of data is done with high performance. The second goal is to provide fast object access and navigation through structures of objects which are connected via logical pointers.

Conventional relational data can be retrieved as usual by passing SQL requests to the liveCache. The object data can only be accessed by special stored procedures which run as part of the liveCache process. This has the advantage that the object data is accessed with high performance because the application and the object data are located in the same address space and therefore no address space switch is needed.

1.1 The liveCache in the R/3 System

Figure 1 shows how the liveCache is connected to the R/3 System. An R/3 work process can be connected to more than one DBMS[2]. With one connection, the R/3 work process is connected to the main RDBMS. Another connection is used for the connection with the liveCache.

RDBMS

The RDBMS normally has a relatively small database cache. The data mainly resides on disk. With this data, it is possible to fully recover the database after a system crash.

liveCache

In contrast to the RDBMS, the liveCache has a large cache (memory persistent data) in which the data normally resides in. In case of a system crash, only the relational data may be fully recovered, the persistent objects are not completely recovered[3]. The ABAP application is responsible for calling the stored procedures which create and store the objects in the liveCache.

[2] The R/3 database interface allows that there are other DBMS connected to the R/3 workprocess beyond the main R/3 DBMS. As of R/3 release 4.6 calls to these additional DBMS can only be passed using native SQL, open SQL is only possible for the R/3 main DBMS.

[3] Modifications on persistent objects between the last checkpoint and a system crash are not recovered. A checkpoint saves the current state of the liveCache data.

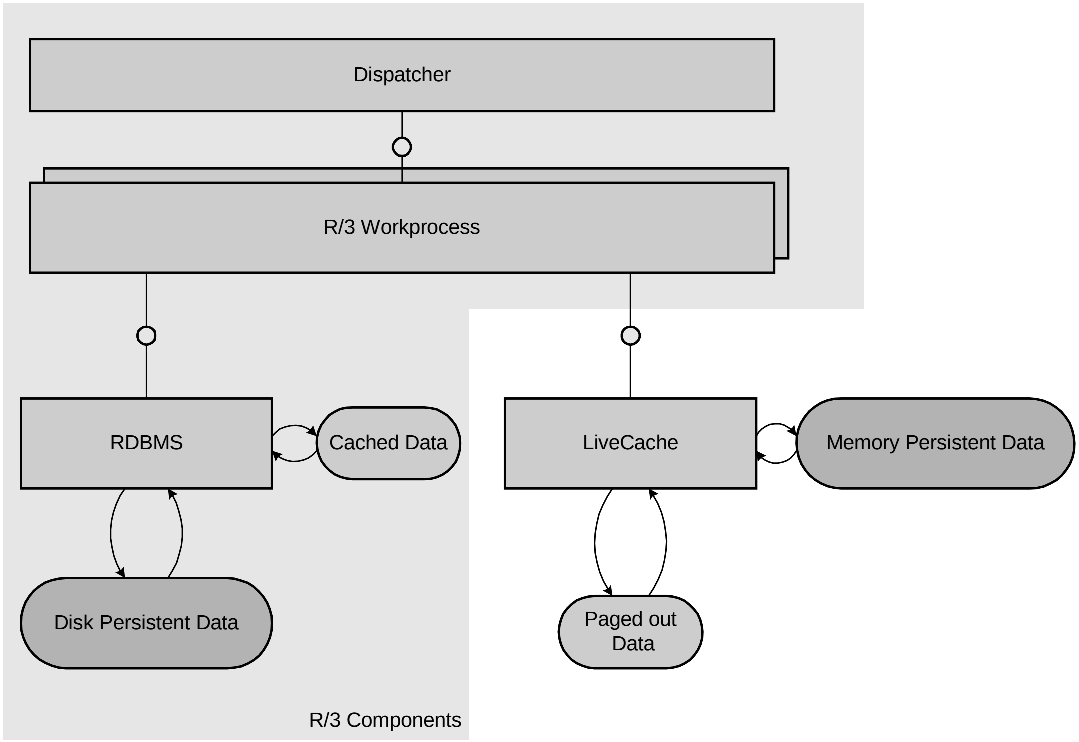


Figure 1 LiveCache and the R/3 System

User Task

Figure 2 shows the connection between the R/3 System and the liveCache in more detail. Each work process is connected to a user task which is running as part of the liveCache and which is a dedicated server for the connection. Requests from the R/3 work process (ABAP application) can only be passed to the liveCache via native SQL; open SQL is not supported yet.

LC Basis Services

The user task includes the request manager and the liveCache basis (LC basis). The request manager receives the requests and controls the execution with the help of the LC basis, which implements the kernel functions of the liveCache. The LC basis services, such as services for disk I/O operations, are shared between the user tasks.

Application Context

The persistent objects need to be accessed by stored procedures while this is optional for SQL data. The advantage of accessing data by stored procedures is that they run in the context of the liveCache process and therefore data access is fast. These stored procedures are implemented as part of special classes which implement the liveCache part of the application logic (e.g. the liveCache part of the APO). In this report an instance of such a class is called application context and its methods which are executed as stored procedures are called application context methods. An application context is that part of an application, which is executed as part of the liveCache process. An example for such an application is the APO (Advanced Planner and Optimizer). From the work process (ABAP application), an application context method can be called via a native SQL statement. The application contexts are created and destroyed by the request manager.

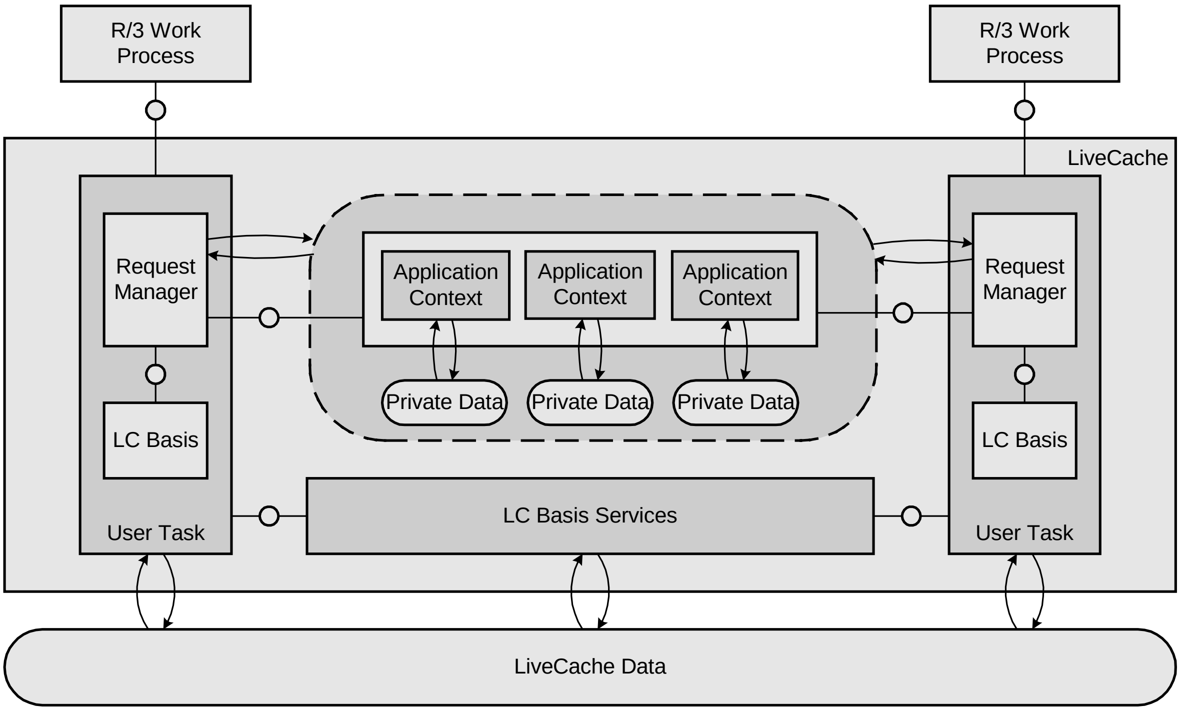


Figure 2 LiveCache and R/3 Work Processes

1.2 General Terms

This section introduces the general terms persistent object, consistent view and version.

Persistent Object

As mentioned above, beyond the SQL data, the liveCache handles the persistent objects. A class of persistent objects is developed in C++ and derived from a special class provided by the object management system. The persistent objects are created, modified and stored by the application context. The liveCache attempts to handle all data in main memory, therefore also the persistent objects reside in main memory.

Private Data

When an application context accesses a persistent object the first time in a transaction, the liveCache creates a copy of the persistent object in the private data of the application context. The copies in the private data can be accessed directly by physical pointers, whereas persistent objects in the LC basis are accessed by logical pointers. Therefore accessing the object copies is much faster than accessing persistent objects in the LC basis. Modifications on persistent objects are applied to the copies in the private data at first. They are stored in the LC basis later, when the transaction is committed. This improves performance, because when accessing a persistent object more than once in one transaction, the persistent object needs only one time to be accessed in the LC basis. This behavior however needs special mechanisms to provide consistency.

Consistent View / Consistent Read

The liveCache provides mechanisms which ensure that several persistent objects read within one period (in general within one transaction) are consistent with each other. All read operations in this period are based on one single consistent state of all persistent objects in the liveCache. This state is called consistent view. A consistent view is a snapshot of all persistent objects in the liveCache in their committed state. Generally the consistent view is created when the first read operation of a transaction is performed and is valid until the transaction ends. The term consistent read describes the process of reading a persistent object in the appropriate state by the LC basis.

Version

Another important concept is the concept of versions. A version is based on a consistent view and has a separate memory area which belongs to the version. Unlike normal processing, when a version is active, modifications are never stored in the liveCache data (neither when processing commit) but only in the separate memory area of the version. A version can be accessed in different transactions but only within one session at a time. A version is a kind of playground were the application can simulate operations first before running them within a real liveCache transaction. The APO uses versions to try out a planning operation first. During the transaction simulation, the APO logs the modifications in a so-called delta queue. If the simulation was successful, the APO runs a liveCache transaction which executes operations based on the delta queue. For a more detailed description of the version concept refer to section 2.3.

1.3 Rough structure of the liveCache

SQL Request Manager

Figure 3 shows the rough structure of the liveCache. At the top of the block diagram there is the liveCache client, which in an R/3 context is an R/3 work process. The liveCache client is connected to the SQL request manager, which is responsible for receiving SQL requests as well as for preparing and controlling the execution of SQL statements.

Session Context

Information about the execution and preparation of SQL requests is stored in the session context. The session context belongs to the connection. It can not be shared between different connections.

OMS Request Manager

The OMS request manager is responsible for the execution of OMS requests with the help of the LC basis. It handles the create, read, store, delete and lock operations on persistent objects as well as other transaction control statements.

Application Context

The application context is implemented by the application programmer with C++ and registered to the liveCache. The call of an application context method is also an SQL statement. When an application context method is called, the request manager checks whether the corresponding application context has already been instantiated. From version 7.2 on, there are application contexts which are created and destroyed by the liveCache client and which are session independent as well as others which are created and destroyed implicitly and bound to the liveCache session. If an application context does not exist, an error for the first kind occurs, the other kind of application context would be created implicitly. The application context method can send OMS or SQL requests to the request manager

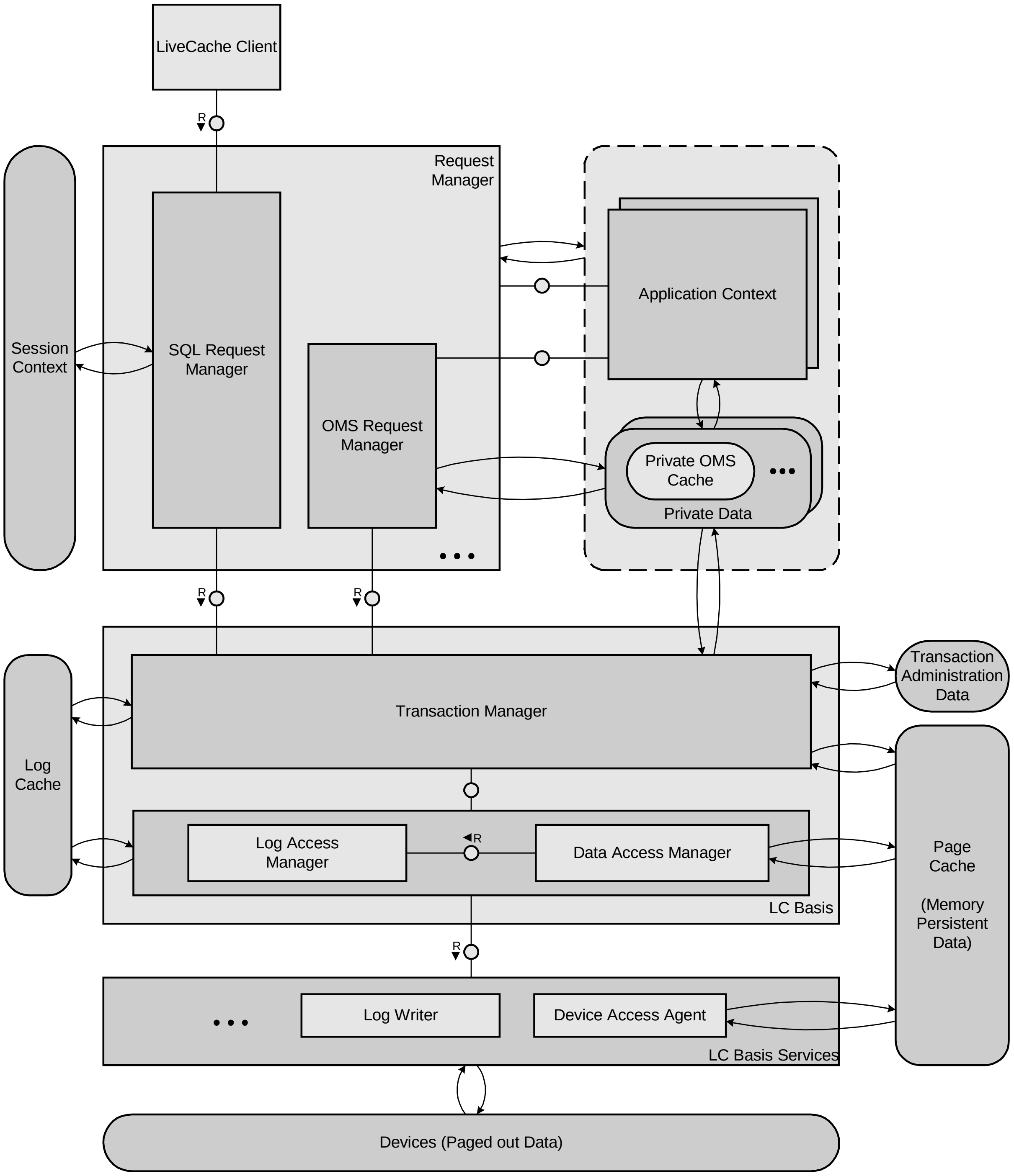


Figure 3 Rough Structure of the liveCache

Private OMS Cache

Bound to the application context is the private data of the application context. The private OMS cache is a part of the private data. Persistent objects which were read out of the liveCache data (device or page cache) are cached in the private OMS cache. When the OMS request manager handles a read request, it first checks whether the corresponding persistent object is already cached in this private OMS cache. If the persistent object does not exist in the private OMS cache, the OMS request manager sends a request to the LC basis, which references the object in the liveCache data and creates a copy in the private OMS cache. When the current consistent view changes[4], the private OMS cache is invalidated. The concept of the consistent view is discussed later in this chapter.

LC Basis

Both, the SQL request manager and the OMS request manager, send requests to the LC basis, which is responsible for the execution. The SQL request manager parses SQL requests and divides them into single steps which are passed to the LC basis. The OMS request manager requests the LC basis in order to access persistent objects.

Transaction Manager

The transaction manager receives the requests addressed to the LC basis and controls their execution. It is responsible for controlling data access, consistent read, logging, locking and EOT[5]-management.

Transaction Administration Data

The transaction administration data includes information about open transactions, lock tables etc. It is maintained by the transaction manager.

Data Access Manager

Page Cache

If data (e.g. persistent objects or SQL data) is to be read or updated, the transaction manager calls the data access manager, which first tries to locate the corresponding data in the page cache (memory persistent data). Generally, the size of the page cache is sufficient[6], so that data is located in the cache. In special circumstances, it is possible that data was paged out and needs to be retrieved from a data device (paged out data).

Device Access Manager

Devices

In this case the data access manager requests the device access agent to copy the requested page from the corresponding device into the page cache. If there is no free memory available, a page must be selected to be paged out. There are different devices for log pages (log devices), SQL and object pages (data devices), as well as for other system data (system devices). Physically, a device is an operating system file on the hard disk.

After the page is located (or is retrieved from a device) the data access manager calls back to the transaction manager. If a read request is invoked, the transaction manager references the corresponding consistent object and creates a copy in the private OMS cache.

As far as reading log entries is concerned, the data access manager first tries to get the log page from the page cache. If this fails, it tries to get it from the log access manager and in the worst case, it reads the page from the log device.

Log Access Manager

If the transaction manager needs to create log entries, it passes requests to the log access manager. The log access manager references the next log position and the transaction manger writes the entry.

[4] This happens at the end of the liveCache transaction, when a version is activated or when a new consistent view is created

[5] <u>E</u>nd <u>O</u>f <u>T</u>ransaction, commit or rollback.

[6] This is a characteristic of the liveCache because it has a large cache.

Log Writer

If a log page is full, the log access manager triggers the log writer to write this log page to the log device. The log writer writes this log page asynchronously. Thus, the log access manager does not have to wait for it. Object log pages are additionally written to the page cache in order to improve performance. Therefore object log pages mainly reside in the page cache. This is the reason why when reading a log page, the transaction manager requests the data access manager first to try to retrieve a log page from the page cache.

Data Storage Areas

The following paragraphs examine where the persistent objects and the SQL data can be located.

Figure 3 depicts four different storage areas where persistent objects and SQL data may reside. These are the private OMS cache, the page cache, the devices and the log cache.

Private OMS Cache

As mentioned above, persistent objects are cached in the private OMS cache, which is private to the application context. If an object is found in the private OMS cache, reading the object out of the liveCache data would result in the same object value[7]. The private OMS cache is only for persistent objects, SQL data may not reside in this storage.

Page Cache

A second storage area is the page cache. This is the main cache of the liveCache where normally all the persistent objects, SQL data and log entries are stored.

Log Cache

The log cache includes the newest log entries. Log entries are first created in the log cache. When a log page is full, the log entries of this page are written to the log device and the page cache. The log cache includes before and after images of SQL data which are needed for database recovery and before images of persistent objects which are needed for the consistent read. As the object data is not fully recovered in case of a system crash, after images of the persistent objects are not created.

Devices

On the devices, the object, SQL and log data is stored. If the size of the page cache is not sufficient, the object and SQL data is paged out to the data devices. Log pages are written to the log device continuously.

1.4 The OMS Classes

The OMS request manager is implemented as a set of C++ classes. Some of these classes provide a generalization of the application contexts, others provide a generalization of the persistent objects. The developer of the application context derives from these classes to get the functionality to create stored procedures and handle persistent objects. In the following, the structure of the OMS classes will be examined. **Figure 4** shows a class diagram of this structure

[7] Provided that the object was not modified in the private OMS cache

The left part of the diagram depicts the classes which form the OMS request manager.

OmsHandle

The class OmsHandle defines the interface between the application context and the liveCache. It implements all methods needed to lock, store, create and delete persistent objects in the liveCache as well as to manage the current OMS transaction. The developer derives the application context from the class OmsHandle and implements the stored procedures as methods of this derived class.

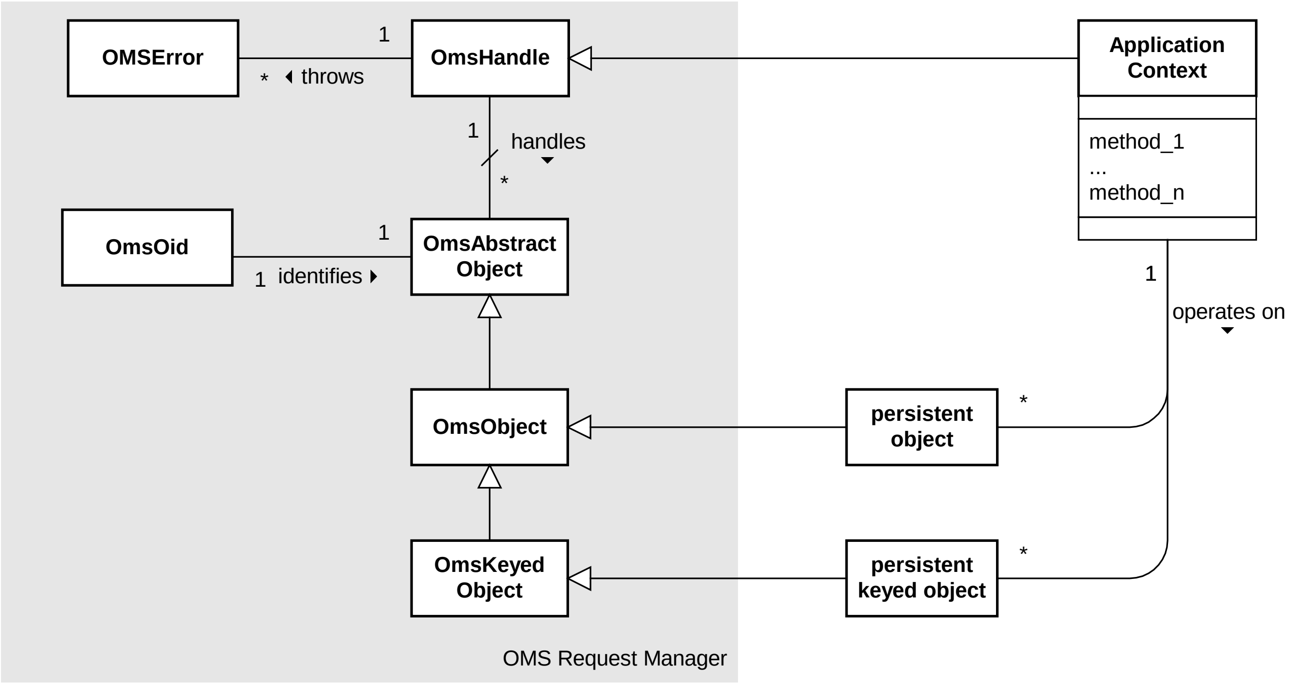


Figure 4 OMS Class Structure

OMSAbstract Object

The class OmsAbstactObject defines the basic structure of a persistent object. It includes the methods to lock, store, create and delete persistent objects. These methods call the corresponding methods from OmsHandle. The association of OmsHandle to the class OmsAbstractObject means that the OmsHandle handles the persistent objects in that way that they are created, stored, locked and deleted. This is not an implemented association, but it results from the derivation of the application context from OMSHandle and the fact that the application context operates on the persistent object.

OmsOid

Each persistent Object is uniquely identified by an object identifier (OID). This identifier is implemented in the class OmsOid. The OID is needed to reference a persistent object in the liveCache.

OmsObject OmsKeyed Object

The class OmsObject defines the methods of a persistent object. It is a specialization of the class OmsAbstractObject and therefore inherits its basic structure. The class OmsKeyedObject defines a special kind of persistent object which can additionally be referenced by a key.

The persistent objects on which the application context operates are derived from OmsObject or OmsKeyedObject. As the application context is derived from OmsHandle and the persistent objects resides in the context of the application context, the “this”-pointer of the application context is passed as parameter when calling a method of a persistent object.

1.5 OMS Transaction Concepts

1.5.1 Consistent View and Consistent Read

A very important concept of the OMS-part of the liveCache is the consistent read. Consistent read in the sense of the liveCache should not be mixed up with Oracle’s implementation of the consistent read. LiveCache consistent read means that all read operations within a period (in general within a transaction) will see the objects in the same committed state they were when the first read operation was processed. This also holds for objects that have been changed by other transactions in the meantime. Oracle’s consistent read always reads the newest committed data. **Figure 5** depicts the logical behavior of the liveCache consistent read.

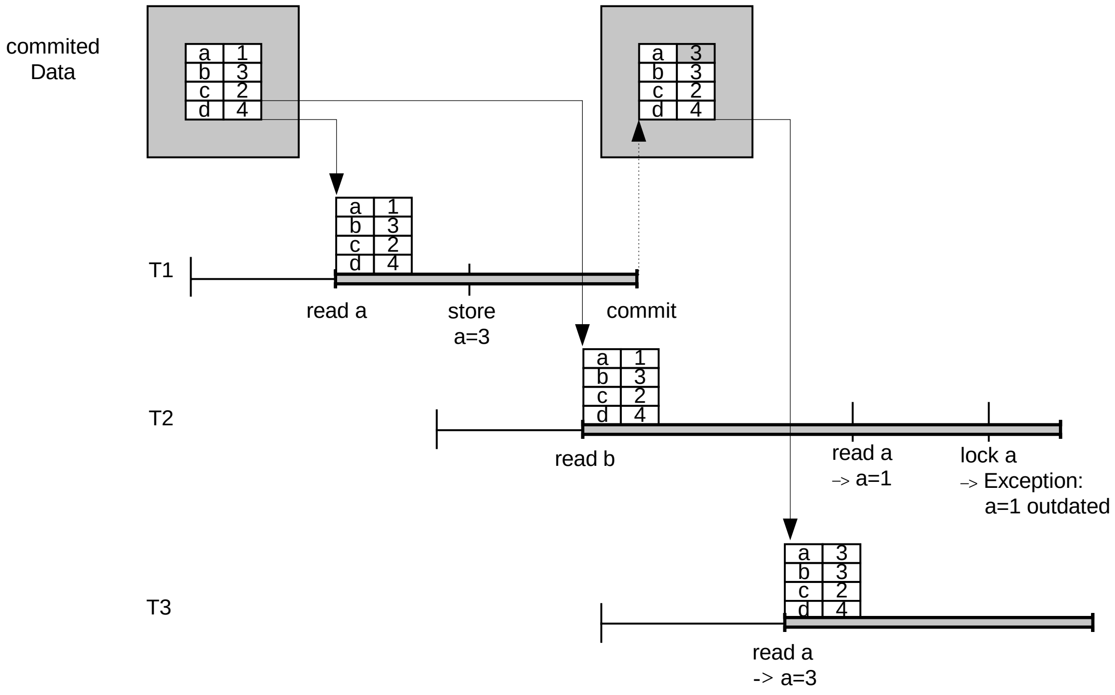


Figure 5 Concept of the liveCache Consistent Read

Logical Behaviour

When a transaction processes its first read operation, the transaction, from a logical point of view, gets a copy of all objects in their committed state[8].

[8] This also happens when a transaction explicitly creates a new consistent view.

This copy forms the so-called consistent view of the transaction[9]. The following read operations work on this virtual copy.
When transaction T2 performs its first read, the modifications of transaction T1 are not yet committed. Therefore the logical copy of the committed data contains the object *a* with the old committed value a=1. When transaction T2 reads object *a*, the modification of object *a* with value a=3 is committed, nevertheless T2 reads the old value (a=1) because the logical copy was created before the commit of T1 and therefore contains object *a* with a=1. This is different to Oracle's concept of the consistent read, which always reads the newest committed data.

Now refer to transaction T3. Transaction T3 reads object *a* earlier than transaction T2, nevertheless it gets the new value of *a*, with a=3, because at the time the first read operation occurs, the modification of transaction T1 has already been committed.

Figure 6 shows effects of the concept of consistent view shown by the example of two concurrently running transactions with different relations in timing. A consistent view can also be created by an explicit call. In Figure 6 it is not distinguished between an implicit consistent view, created at the first read of a transaction, or an explicit consistent view.

Figure 6a

In Figure 6a, transaction T2 creates the consistent view after transaction T1 has committed. Therefore it reads the new value of object *a*.

Figure 6b
Figure 6c

In Figure 6b, transaction T2 reads the old value of object *a* because Transaction T1 is not yet committed. The same holds for transaction T2 in Figure 6c with the difference that in Figure 6c, transaction T1 is not yet started when transaction T2 creates its consistent view.

Figure 6d
Figure 6e

In Figure 6d the changes of transaction T1 are committed when Transaction T2 reads object *a*. Nevertheless, transaction T2 reads the old value a=1, because it is not relevant whether changes are committed at the time the read operation occurs, or at the time when the consistent view was created. This is the same with Figure 6e.

Implementation

Obviously, the consistent view is not implemented by copying all the objects completely. The realization of the consistent read uses the before images in the log. The log includes the history of changed objects, which means that one object can possibly reside in the object log more than once, each time with a different state. A detailed description of how the consistent view is implemented will follow in section 3.2.1 of this Bluebook.

[9] In general the consistent view exists from the first read operation until the end of the transaction, but it is also possible to explicitly create a consistent view during a transaction. This causes the current consistent view to be destroyed. For simplicity, it is assumed in this example that there is exactly one consistent view for a transaction.

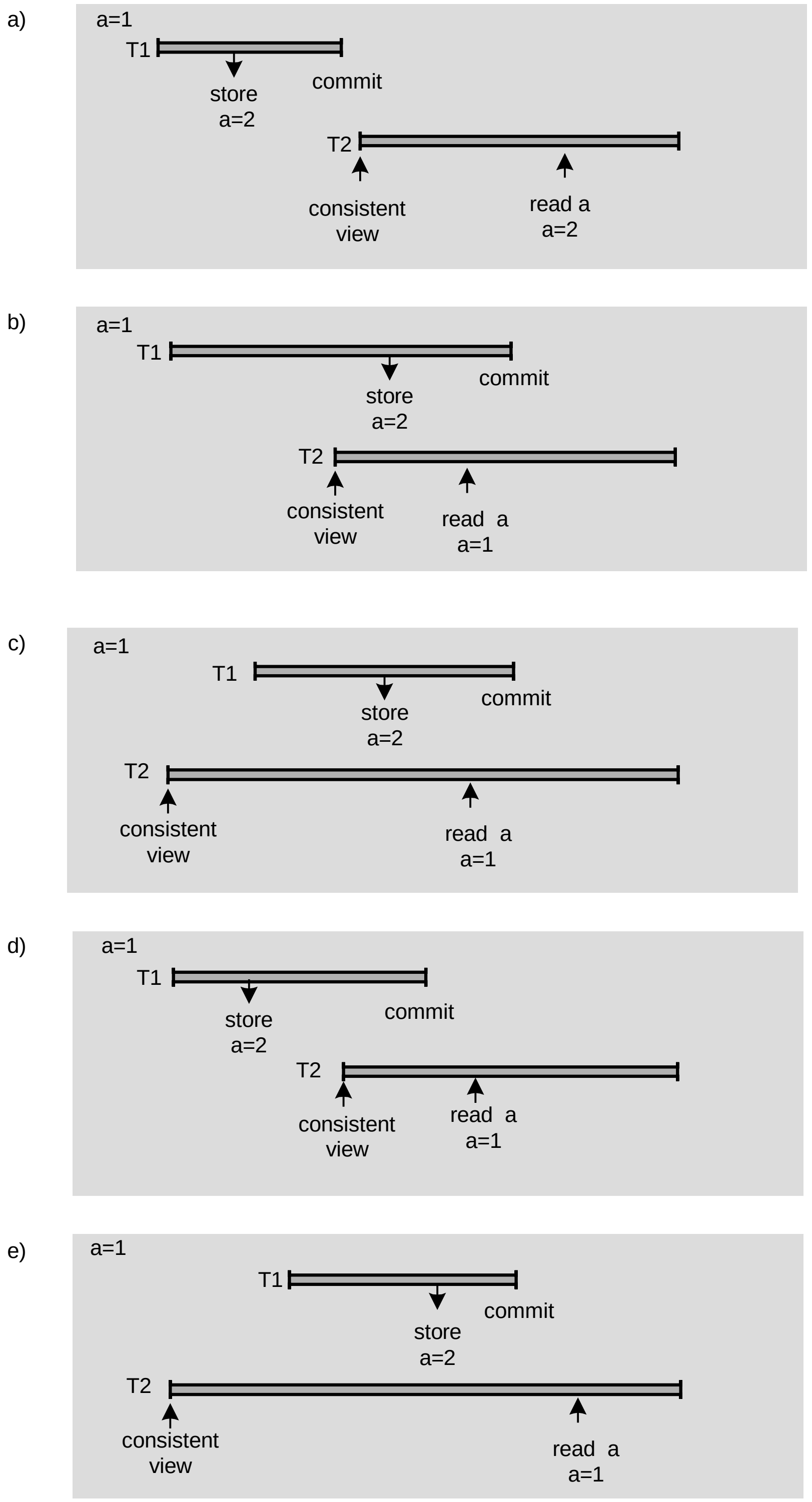


Figure 6 Dependencies between two concurrently running Transactions

1.5.2 Locking

Locks

To isolate concurrent transactions, the OMS uses exclusive locks. A persistent object can only be changed if the corresponding transaction owns an exclusive lock on this object. Locks are requested explicitly by calling the corresponding method of the persistent object. The OMS does not know shared locks.

Getting Locks

A necessary requirement to get a lock on a persistent object is that no other transaction owns a lock on the same object. However, this is not sufficient. It is possible that a persistent object has changed and the change has already been committed, so that the object is not locked anymore. Consistent read can cause a transaction to still own an old view on this object. The OMS has to guarantee that the updated object is not overwritten by the old view. Therefore, it is not only checked if the object is locked by another transaction, but also if the object has changed since the beginning of the consistent view. In Figure 5, transaction T2 shows an example of this fact. The consistent view of transaction T2 sees the old state of object *a*. In the meantime, transaction T1 has committed and when transaction T2 wants to lock object *a*, it can not get the lock, because object *a* has changed since the beginning of the consistent view.

Lock Lists

A particular kind of locking is to request locks on several persistent objects at once. For this, a list of these objects is passed to the OMS request manager which forwards the requests to the liveCache basis. The requesting task waits until all locks are granted or a timeout is reached. After getting the locks, a new consistent view is started. For this kind of locking it is not required to check if the object has changed since creating the consistent view, because a new consistent view is started after getting the locks.

1.5.3 Update Processing

The OMS request manager does not pass update requests to the transaction manager immediately. Instead, updates are marked in the private OMS cache. During commit processing these updates are sent to the transaction manager which processes the store operations. Lock requests are directly passed to the transaction manager. This ensures that during commit processing the updates can be processed successfully with the assumption that the corresponding objects are locked beforehand. From the user’s point of view, this process is atomic. It is executed entirely or not at all in case of an error. An error in this context means that the transaction does not own a lock on an object which is to be updated[10].

1.5.4 Sub Transactions

The liveCache provides sub transactions. Sub transactions are used in order to divide the whole transaction into a hierarchy of sub steps which form

[10] In release 7.1 the liveCache provided three different transaction modes. The described behavior was known as mode “pessimistic”. Beyond this, the liveCache provided the modes “optimistic” and “implicit”. In mode “optimistic” also locks were marked in the OMS cache. The disadvantage was, that other transactions could modify the objects and therefore commit processing often failed with exception “outdated”. In mode “implicit”, locks were requested implicitly by reading an object. This mode was very poor in sense of limited concurrency.

logical coherent units of the transaction. A rollback or commit on a sub transaction only affects modifications which occurred in the sub transaction itself or in sub transactions below this sub transaction. Even if a sub transaction has committed, the parent of the sub transaction can still rollback this sub transaction[11].

In general, when talking about commit or rollback in this report the commit or rollback on the main transaction is meant.

[11] One should not confuse sub transactions with R/3 transactions started with the statement “call transaction”. An R/3 transactions started with “call transaction” is inserted into the current transaction but the calling transaction can not control commit or rollback on the called transaction. This is different to sub transactions.

2. Client Communication Layer

The architecture of the liveCache can be divided into two layers. These are the client communication layer, which implements the top-level components of the liveCache, and the liveCache Basis, which implements the liveCache kernel functions and structures.

This chapter describes the architecture of the client communication layer. The client communication layer is responsible for receiving and preparing the SQL and OMS requests, as well as for controlling their execution by calling the liveCache basis layer. However, some OMS requests do not affect the liveCache basis layer, because they are only relevant for the objects which are cached in the private OMS cache.

2.1 Components of the Client Communication Layer

Figure A1 shows the components of the client communication layer. While Figure 3 describes the rough liveCache structure, Figure A1 shows the structure in more detail. Furthermore, the OMS SQL interface, the EOT handler, the application context manager and the callback interface are added in Figure A1.

SQL Request Manager

The SQL request manager is divided into the client communication manager, the command analyzer and the SQL controller. The client communication manager is responsible for receiving the client requests and for sending the response to the client. The command analyzer analyzes the SQL commands and calculates the execution plan. The SQL controller controls the execution of SQL requests and triggers the application context manager to start application context methods.

Application Context

The application contexts contain application context methods. They implement the part of the application (e.g. the APO) which is executed in the context of the liveCache kernel process. They reside in the same address space as the liveCache data (persistent objects and relational data) and can therefore access the data with high performance. There are two types of application contexts, the session dependent and the session independent ones, which differ in lifetime. While the lifetime of a session dependent application context is restricted to the lifetime of the liveCache session, the lifetime of a session independent application context is controlled by the liveCache client. If a session independent application context is not destroyed, it exists until the liveCache is restarted. Another difference is that a session dependent application context is bound to a liveCache session while a session independent application context can be moved between different liveCache sessions. If a session independent application context is used in a liveCache transaction, it can not be used by other transactions until the using transaction has terminated.

Private Data

Bound to the application context is the private data. Part of this private data is the private OMS cache, the local heap and the local data. The private OMS cache includes the copies of the persistent objects. This is the part of the private data which is controlled by the liveCache. The local data and the local heap are only controlled by the application context except that the OMS request manager stores data which is needed to rollback a sub transaction on the heap. The liveCache has no knowledge about the other contents in these data areas.

Application Context Manager

The application context manager is responsible for calling the application context methods. It receives the corresponding requests from the SQL request manager. Furthermore, it controls the lifetime of the application contexts.

OMS SQL Interface

The OMS SQL interface may be used by the application contexts to pass SQL requests to the liveCache. It is implemented as a C++ class. By deriving the application context from the OMS SQL interface, it inherits the functionality to create SQL queries.

EOT Handler

Callback interface

The EOT handler is responsible for handling a commit or rollback on the client application layer. There are several tasks which must be performed by the EOT handler in case of an EOT request, e.g. the application contexts have to be informed about the commit or rollback via the callback interface and the objects modified in the private OMS caches have to be stored in the liveCache data.

OMS Request Manager

The OMS request manager is divided into the OMS interface, the sub transaction handler, the version manager and the object access agent. The sub transaction handler handles sub transactions and the logging on the client communication layer. Logging on this layer is needed, because modifications which occur in a sub transaction affect only the private OMS cache. In case of a sub transaction rollback these modifications have to be undone. The version manager handles operations on versions and controls the object access while a version is active. Furthermore, it handles the requests to explicitly create a new consistent view. The object access agent handles the create, store, delete and lock requests on persistent objects.

2.2 Private OMS Cache

Each application context has its own private OMS cache. On a read request on a persistent object, the object access agent first checks if the object is already in the private OMS cache belonging to the application context which created the request. The read request is only forwarded to the liveCache basis layer if the requested object can not be found in the private OMS cache. Furthermore, store operations only affect the object copies in the cache until the transaction is committed[12].

[12] In the special case of objects with variable length, no object copies in the private OMS cache are created and store operations are directly passed to the liveCache basis.

Private OMS Cache

Figure 7 shows the structure of the private OMS cache. The persistent objects are organized by a hash table, which maps the OID to the beginning of a collision list. The elements of this list are the persistent objects with additional administrative data containing the OID, the sub transaction which last modified the object (sub ta), the before reference (bef ref) and a state byte.

Before Reference

The before reference points to the before image of the object in the liveCache basis. When locking an object, the liveCache basis uses the before reference in order to check if the object was changed since the beginning of the consistent view. If the before reference of the original object in the liveCache basis differs from the before reference in the object copy of the private OMS cache, the object has been changed by another transaction since the consistent view was started and a lock can not be granted.

State Byte

The state byte includes the delete flag, the write flag and the lock flag. The delete and the write flag determine that the object was modified in the private OMS cache and that a delete or store request respectively has to be passed to the LC basis in case of commit. The lock flag determines if the current transaction owns a lock on the object. It is used for optimization. If the transaction already owns a lock, a second lock request on the same object would be performed very fast. If the flag is set, the transaction definitely owns a lock. Otherwise, no information about the lock state is available.

The object body includes the data of the persistent object.

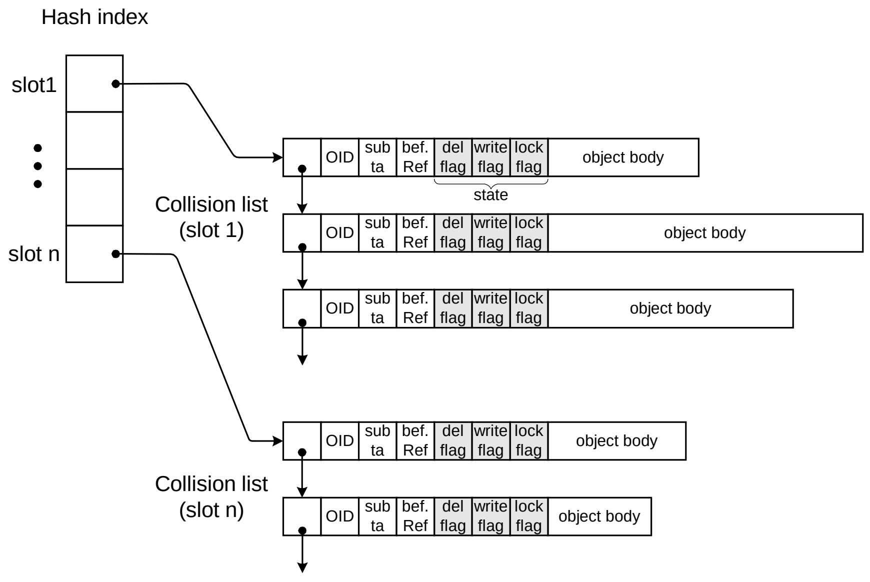


Figure 7 Data structure: Private OMS Cache

At the time of writing, it is possible to access more than one application context within a transaction. Note that this can cause inconsistencies. When in an application context method, a persistent object is read and modified in the private OMS cache, this modification is not stored in the liveCache data and the private OMS cache of other application contexts. Therefore the modification is not visible for other application contexts. Accessing the same persistent object, within the same transaction, in a different application context can result in a different object value. In further releases of the live-

Cache it may be that a transaction is restricted to access only one application context.

2.3 OMS Request Manager

OMS Interface

In this section the OMS request manager is described with its tasks. The OMS interface is the part of the OMS request manager which receives the OMS requests. For a sub transaction rollback or commit it forwards the request to the EOT handler, for accessing a persistent object to the object access manager and for accessing a version to the version manager.

Object Access Agent

The object access agent is the main component of the OMS request manager. The following describes the different responsibilities of the object access agent.

Reading Objects

When the object access agent receives a read request, it first checks the private OMS cache. It uses the OID to calculate the slot of the hash index. If the requested object can be found in the collision list, a reference to this object is returned. Otherwise, a request is passed to the LC Basis, which copies the object into a specified memory area in the private OMS cache. Subsequently, the object access agent updates the hash table which manages the private OMS cache and returns the object reference.

Lock Request

Lock requests are handled as follows: If a version is active, a lock request is successful in any case, because the version data is private. If the lock flag of the object in the private OMS cache is already set, the lock request is also successful because the lock has been granted already. Otherwise, the lock request is directly passed to the LC basis. The LC basis needs the before reference in order to check if the object has been changed by another transaction since the consistent view of the current transaction. In this case the object would be outdated and therefore a lock can not be granted. If the lock could be requested successfully, the lock flag of the corresponding object in the private OMS cache is set. If the particular kind of locking, where locks on several objects at once are requested is used, a list of OID’s[13] is passed to the OMS request manager. For each entry in the list the lock requests are passed one after another to the LC basis. These locks are requested with a timeout. The task waits until the lock is granted or the timeout is reached. If a timeout is reached, the OMS request manager releases the locks requested so far and raises an exception. Otherwise, after successfully locking all objects belonging to the list, a new consistent view is started. For this kind of lock, the LC basis does not need to check if the object has been changed since the consistent view.

Storing Objects

As mentioned above, an object modification is not stored in the liveCache data until the whole transaction is committed. It is only stored in the private OMS cache at first and the corresponding write flag is set.

[13] To avoid dead locks with locks on objects from this list, the caller (application context) sorts the list in a way that such dead locks do not occur.

Creating Objects

When the object access agent is requested to create a new persistent object, it passes the request directly to the LC basis together with the ID of the class the object belongs to and a reference to the data of the new object in the private OMS cache. The LC basis creates the new object and passes the OID back to the OMS request manager. Finally, the OMS request manager updates the hash table in the private OMS cache.

Deleting Objects

For a delete request, the delete flag of the corresponding object in the private OMS cache is set. The delete request is passed to the LC basis when committing the transaction.

Version Manager

The version manager is responsible for the version management and for handling the requests to explicitly create a new consistent view. In the following the version concept is described in more detail. A version is a kind of playground where first modifications can be tried out. Modifications which are stored while a version is active never affect the liveCache data. From a technical viewpoint, a version is a special consistent view, with the difference, that it can be activated in different sessions and that modifications are never stored in the shared liveCache data. A version has a special memory area named version data (refer to Figure A1) which belongs to the version. When committing a transaction while a version is active, the store requests are not passed to the LC basis as usual. Instead, the modifications are stored in the version data of the version. Locks on the persistent objects are not needed while a version is active.

When an application wants to execute operations which were first tried out as part of a version in a “real” liveCache transaction later, it is responsible itself to log the modifications beforehand. The liveCache does not provide any mechanisms to log the operations which are performed while a version is active.

APO and Versions

The APO for example uses versions to first try out a planning operation and only executes it as a “real” transaction when successful. While trying out the planning operation, the APO logs the actions in a so-called delta queue. The planning operation can cover more than one R/3 dialog step. Different dialog steps run in different liveCache transactions and generally on different liveCache sessions[14]. Therefore, at the end of a dialog step, a version is closed and in a new dialog step the corresponding version is activated again. When finishing the planning operation, the APO requests a new consistent view and locks all persistent objects it wants to modify. Finally it executes the planning operations according to the delta queue.

Operations on Versions

There are four operations on versions: create, drop, open and close version. Create version saves the current consistent view as a version and activates it. With open version a version is activated, provided that it is not active in another session. With close version the current version is left and a new consistent view is started. Drop version deletes the version and frees up all resources which are used by the version. A version exists until it is explicitly dropped. It is important to drop a version when it is not needed any longer,

[14] This is because different dialog steps are generally executed within different R/3 work processes and each work process is connected to its own liveCache session.

because the LC basis has to provide the consistent view belonging to the version until the version is dropped. The version manager is responsible for the version management.

Version vs. Consistent View

There are some distinctive features which appear when accessing a persistent object while a version is active. For a read request on an object which does not already exist in the private OMS cache, the object access agent first requests the object from the LC basis layer. After this, it is checked if a modification on this object is already stored in the version data. In this case the object from the version data, otherwise the object from the LC basis is returned.

Lock requests are always successful because the version is private and therefore a lock is not necessary. When a version is active, locks have no effect.

Version Data

The persistent objects which are stored while a version is active are stored in the version data. The version data is structured as a B* tree[15] and the persistent objects are stored as records in this B* tree. The OID of a persistent object forms the key part, the object data forms the non key part of a record stored in the B* tree.

New Consistent View

Another task of the version manager is to handle the requests to explicitly create a new consistent view. For this request, the version manager invalidates the persistent objects in the private OMS cache of the application contexts involved[16] and requests the new consistent view from the LC basis. Objects on which the transaction owns a lock are not invalidated since they can not be changed by other transactions.

Sub Transaction Handler

The sub transaction handler manages the sub transactions on the client communication layer. Requests concerning sub transactions are only received from the application context. The liveCache client can not control sub transactions directly via SQL. Note that sub transactions are only valid within the application context which started the sub transaction. When leaving an application context method, all sub transactions created in the corresponding application context should be closed. Sub transactions are mainly processed without the LC basis[17]. For the request to start a sub transaction, the sub transaction handler saves all persistent objects which were modified on the local heap. These copies are used by the EOT handler in case of a sub transaction rollback in order to recover the state of the OMS cache which was valid before starting the sub transaction. Furthermore, the sub transaction handler modifies the transaction management data, which includes information about the sub transaction currently running.

[15] In this Bluebook a B* tree is regarded together with the data pages which include the data records rather than only as the index.

[16] The involved application contexts are all session dependent application contexts belonging to the current session and all session independent application contexts which were accessed in the current transaction.

[17] The only part of sub transaction handling performed by the liveCache basis is the handling of modifications on variable length objects, because objects with variable length are not stored and modified in the private OMS cache, but directly in the liveCache basis.

2.4 EOT Handler

The EOT handler has two functions. Firstly, it processes the rollback and commit of sub transactions. Secondly, it prepares processing commit and rollback of the whole transaction and requests the LC basis. While the request to commit or rollback a sub transaction can only be received from the application context (via the sub transaction handler), the EOT request for the whole liveCache transaction can only be received from the SQL controller. No request from an application context is possible which may commit or rollback the whole transaction. If the application context sends a corresponding request using the OMS SQL interface, the request will be discarded.

Callback

The EOT handler passes messages to the application context via the callback interface. When the main transaction is committed, the application contexts involved (refer to footnote 16, page 22) are informed before processing the commit in order to prepare themselves. The callback is necessary, because either the local heap of the application context has to be cleaned up or modifications which are stored in the local data or on the heap have to be stored as persistent objects. Furthermore, the application contexts are informed after commit processing in order to invalidate their local data and heap. The EOT handler can only invalidate the private OMS cache, because it does not know the internal data structures of the local heap of an application context. The invalidation of the local data and heap has to be performed by the application context itself. In case of rollback on the main transaction, the application contexts are also informed after the processing in order to invalidate their heap and local data.

Furthermore, a message is sent before a sub transaction commit and after a sub transaction rollback.

Main Transaction Commit

When committing the main transaction besides calling the involved application contexts, the EOT handler has to pass the store and delete requests which are saved in the private OMS cache to the LC basis[18]. If the operations could be proceeded successfully, the commit is passed to the LC basis layer and the private OMS cache is invalidated.

Committing the main transaction is processed in the following steps:

In a first step, the EOT handler requests the application context manager to loop through all application contexts involved. For each such application context, the application context manager calls the EOT handler in order to process the commit for the application context.

If the application context is registered to the callback interface, the EOT handler first calls the application context via the callback interface. After returning from this call the EOT handler passes all store requests out of the private OMS cache belonging to the application context to the LC basis layer[18].

[18] When a version is active, the store requests are not passed to the LC basis. In this case the modifications are stored in the version data.

If this process was successful for all application contexts concerned, a commit request is passed to the LC basis layer in order to commit the liveCache transaction. Otherwise the liveCache transaction is rolled back.

In a second step, the application context manager loops through the involved application contexts again in order to invalidate their OMS caches. If the application context is registered to the callback interface, the invalidate method is called in addition. Finally a new consistent view is started.

Main Transaction Rollback

For a main transaction rollback, the EOT handler requests the LC basis to rollback the liveCache transaction. On return, the private OMS cache is invalidated for each application context concerned and, if implemented, the application context is informed via the callback interface. Finally a new consistent view is started.

Note that if a version is active, the EOT on a main transaction does not cause the invalidation of the OMS cache and the creation of a new consistent view. For this case the version will remain active.

Sub Transaction Rollback

Sub transactions are mainly processed without involving the LC basis. In case of a sub transaction rollback, the EOT handler first deletes all persistent objects in the private OMS cache which were modified in the current or lower level sub transaction. Furthermore, it copies the images of the persistent objects, created on the local heap before starting the sub transaction, back to the private OMS cache. The rollback of modifications on variable length objects is performed by the liveCache basis because variable length objects are not stored in the private OMS cache. Persistent objects which were copied to the private OMS cache during the sub transaction are invalidated in the private OMS cache. When accessing such an object after the rollback, the object is copied from the liveCache basis again. Finally, the sub transaction is terminated by modifying the transaction management data and the application context is informed via the callback interface. Note that a rollback on a sub transaction does not release locks which were requested during the sub transaction.

Sub Transaction Commit

For a sub transaction commit, the application context is also informed via the callback interface and may proceed further OMS requests which still belong to the current sub transaction. After returning from this call, the current sub transaction is terminated.

2.5 Application Context Manager

The application context manager is responsible for the instantiation, destruction and administration of the application contexts. Furthermore, it handles the calls of the context methods.

As mentioned above, there are two kinds of application contexts: The session dependent and the session independent ones. The former are instantiated and destroyed automatically. The latter however are explicitly created and destroyed by the liveCache client.

Explicit Creation

If the client wants to create a session independent application context, it passes a SQL request via the client communication manager to the command analyzer, which forwards it to the SQL controller. The client specifies an ID for identifying the application context. The SQL controller requests the application context manager in order to create the application context. If the ID is not valid, an error is raised.

Implicit Creation

On first access the session dependent application contexts are created implicitly. When calling a method, it is checked if the corresponding application context is already instantiated. If necessary it is created.

Instantiation

The instantiation of an application context is processed along the following steps: In case of a session independent context, it is checked if the ID, specified by the client, does already exist. In this case an error will be raised. Furthermore, it is checked if a factory object for this class of application contexts does already exist in the liveCache. The factory object is instantiated when the first application context of a specified class is created and is not destroyed until the liveCache is shut down. The factory object handles the instantiation of application contexts of a specific class.

A session independent context is bound to the ID specified by the client, the session dependent one is bound to the session.

Destroying Application Contexts

If a session independent application context is to be destroyed, the request is also received by the application context manager. If the application context belonging to the specified ID exists, the application context manager calls the destructor of the corresponding application context and deletes it from the instance administration data. Session dependent application contexts are destroyed implicitly when the session is terminated. In this case, the application context manager searches all application contexts which belong to the liveCache session and calls their destructor.

Calling Application Context Methods

A method of a session dependent application context is called in the following way: First, the application context manager checks the catalog data in order to get the classId of the corresponding application context. Second, it reads the instance administration data in order to check if an application context for this liveCache session already exists. If necessary, the context is created. Finally, the corresponding method is called.

For calling a method of a session independent application context, the liveCache client specifies the ID of the application context. The application context manager checks the instance administration data on whether the application context belonging to the ID exists. If it does not exist, an error is raised. Otherwise the application context manager checks if the application context is in use by another transaction which is still open. In this case the application context is not accessible until this transaction has terminated and the application context manager raises an error. If the application context exists and is not in use, the requested method is called.

3. liveCache Basis

This chapter describes the liveCache basis layer. The liveCache basis layer implements the kernel functions to implement transaction management, logging, locking, consistent read and data access. There are different components which execute the SQL- and OMS requests.

3.1 Components of the liveCache Basis

Figure A2 shows the components of the liveCache basis layer. Compared to Figure 3, the components are shown in more detail here. In this section the components are roughly described while a detailed description will follow in the following sections.

The transaction manager, which receives its requests from the client communication layer, or, to be more precise, the SQL-, respectively the OMS request manager, is divided into SQL basis, OMS basis and EOT manager.

SQL Basis

The SQL basis handles the SQL requests received from the SQL request manager. It controls the execution of prepared SQL statements, as well as the SQL logging, SQL locking and the handling of the SQL part of the transaction management.

OMS Basis

The OMS basis handles the OMS requests received from the OMS request manager. It is responsible for consistent read, OMS locking, OMS logging, storing and creating persistent objects, as well as for handling the OMS part of the transaction management.

EOT Manager

The EOT manager is called when a transaction is terminated by commit, respectively rollback. The EOT manager executes all necessary operations on the LC basis layer to commit or rollback a transaction. Note that the EOT manager processes the EOT on the LC basis layer, while the EOT handler is responsible for EOT handling on the client communication layer.

Data Access Manager

The data access manager, which is responsible for the data retrieval, is divided into the data access agent, the BD lock manager, the page access agent, the page lock manager and the LRU manager.

Data Access Agent

The data access agent handles requests to access data in the page cache. Depending on the request type, the SQL access agent, object access agent or the log history access agent computes the logical page number of the page where the corresponding data is stored. Furthermore, it requests a reference to this page in the page cache. For accessing SQL data, more than one page must be retrieved because the first pages include the information on how to navigate through the storage structure (B* Tree) to find the requested page. For object or log history requests, only one page must be retrieved, because the page number is part of the object or log entry ID and therefore known.

Page Access Agent

The page access agent is responsible for referencing a requested page in the data cache. It is divided into the page request manager, the cache access manager and the paging manager.

Page Request Manager

The page request manager controls the page retrieval from the data cache. First, it sends the request to the cache access manager, which, supported by the cache access administration data, checks, if the page can directly be retrieved from the page cache. If it can, the cache access manager returns the reference to this page back to the page request manager. Otherwise the cache access manager requests the paging manager, which returns a reference to a page selected for paging. If the selected page is modified, the page request manager requests the device access agent to write out the selected page first. Subsequently, the device access agent is called to read the requested page to the page cache and finally the page reference is returned to the data access agent.

LRU Manager

The pages are linked in LRU[19] order. This means that the most recently used pages are in the front part of this LRU list and the last recently used pages are at the end. The paging algorithm uses this list to decide which page is to be paged out. The LRU manager is called by the cache access manager and reorganizes the LRU list after a page was accessed.

Page Lock Manager

The access to the data structures (page chains for objects and B* trees for SQL data) must be synchronized. For page chains, this synchronization is implemented by the page lock manager. Beyond this, the page lock manager synchronizes access to single pages.

BD Lock Manager

The BD lock manager is the component which handles the synchronization on B* trees. It is called by the data access agent. The lock information is stored in lock lists. The assignment of a B* tree to its corresponding lock list is stored in the lock list access administration data.

Log Access Manager

The log access manager is divided into the SQL log access manager and the object log access manager, which are responsible for accessing the SQL respectively the object log. Furthermore, in comparison with Figure 3, the SQL and object log queue are added. Full log pages are placed into the SQL, respectively the object log queue. The log writer, splitted into the object log writer and the SQL log writer in Figure A2, asynchronously writes the content of the corresponding log queue to the log file. In case of commit, also a not full SQL log page is written to the log file. In this case, the writing is processed synchronously. The object log writer also writes object log pages to the page cache by requesting the data access agent.

LC Basis Services

The LC basis services include the device access agent for the disk I/O, the data writer for asynchronously writing changed pages to the data device, the log writer for writing out the log pages and the garbage collector for deleting unused pages and object frames.

[19] Least Recently Used

3.2 Transaction Manager

In this section, the transaction manager is examined in more detail. Only the OMS tasks are discussed. As mentioned above, the main tasks of the transaction manager, to be more precise, the OMS related tasks, are the consistent read, the lock management, logging, object modification and the EOT management. In this section the transaction is examined from the viewpoint of the LC basis. Do not confuse the requests on the client communication layer with the requests on the LC basis layer. E.g. a “store” request on the client communication layer is “buffered” and sent to the LC basis when committing the transaction, on the LC basis layer the store request is received from the client communication layer and executed immediately.

3.2.1 Consistent Read

The Introduction already discussed the concepts of the consistent read without focusing on the implementation. This paragraph will explain how the consistent read is implemented.

Database

Figure 8 shows a first example of the consistent read with focus on the implementation details. The top of the diagram shows the different states consisting of the database and the before images. In the example the database only includes four objects *a*, *b*, *c* and *d*. Objects are stored in object frames. Simplified, an object frame consists of a reference to the before image of the last update (beforeRef), the object ID (obj)[20], the object value (val) which forms the object state and the transaction (T_{upd}) which performed the last update on the object.

Before Images

Simplified, a before image consists of the reference to the before image of the current before image (beforeRef), the value (val) of the before image and the transaction which wrote this value (T_{upd}).

Transaction ID Consistent View ID

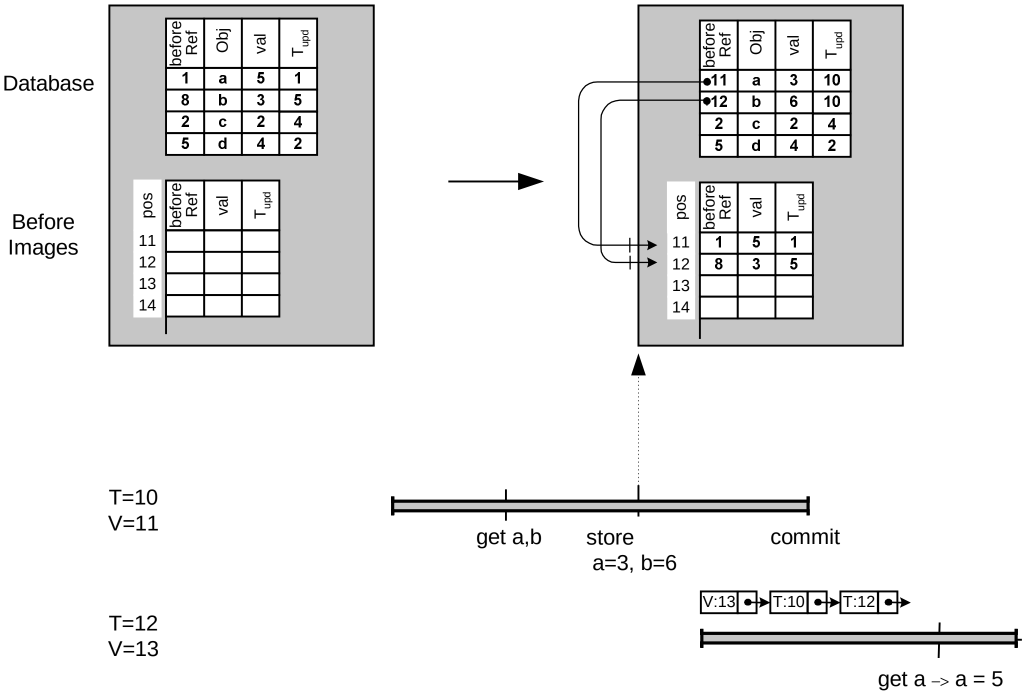
The bottom of the diagram shows the concurrently running transactions. On the left of each transaction, the transaction ID (T=ID) and the ID of the consistent view (V=ID) are shown. The transaction ID identifies the transaction and the consistent view ID determines the consistent view in which the transaction is currently running. The transaction ID is generated when the transaction is started, the consistent view ID when a new consistent view is created. This may happen by the first read operation of the transaction or by an explicit call. For reasons of simplicity, it is assumed that the consistent view is created at the beginning of the transaction and does not change. The two kinds of IDs are generated sequentially from the same pool of sequence numbers. Therefore it is possible to compare the time a transaction started with the time a consistent view was created by comparing the IDs.

Openlist

When a consistent view is created, also the list of currently open transactions is created (refer to the beginning of transaction T=12 in Figure 8). The

[20] The object ID is not really implemented as part of an object frame. It is a logical ID here. In reality, an object is identified by its storage location (page number and offset).

openlist relates the transactions which are open at the time a consistent view is created to the consistent view. In **Figure 8** the consistent view V=13 of transaction T=12 is related to transaction T=10 and T=12 because they are open from the point of the consistent view V=13. The openlist is used in order to verify if an object value is committed and therefore consistent from the point of the consistent view.



before Ref	Obj	val	T_{upd}
1	a	5	1
8	b	3	5
2	c	2	4
5	d	4	2

pos	before Ref	val	T_{upd}
11			
12			
13			
14			

before Ref	Obj	val	T_{upd}
●11	a	3	10
●12	b	6	10
2	c	2	4
5	d	4	2

pos	before Ref	val	T_{upd}
11	1	5	1
12	8	3	5
13			
14			

Figure 8 Consistent Read (example 1)

The following explains why and how transaction T=12 reads the old value a=5 when reading object *a*.

When transaction T=12 reads object *a*, the update of transaction T=10 is already committed. But it was not committed when T=12 started the consistent view, therefore T=12 reads the old value of object *a*.

The read operation is processed along the following steps:

Check Update Time

The transaction manager reads the current object frame of object *a* from the database. First, it checks if the object was changed by a transaction which started after the current consistent view was created. In this case the current object frame would not fit to the consistent view. The check is performed by comparing the current consistent view (V=13) to the ID of the transaction which wrote the update (T_{upd}=10). In this example, the transaction T_{upd} was started before the consistent view was created because T_{upd} =10 is less than V=13.

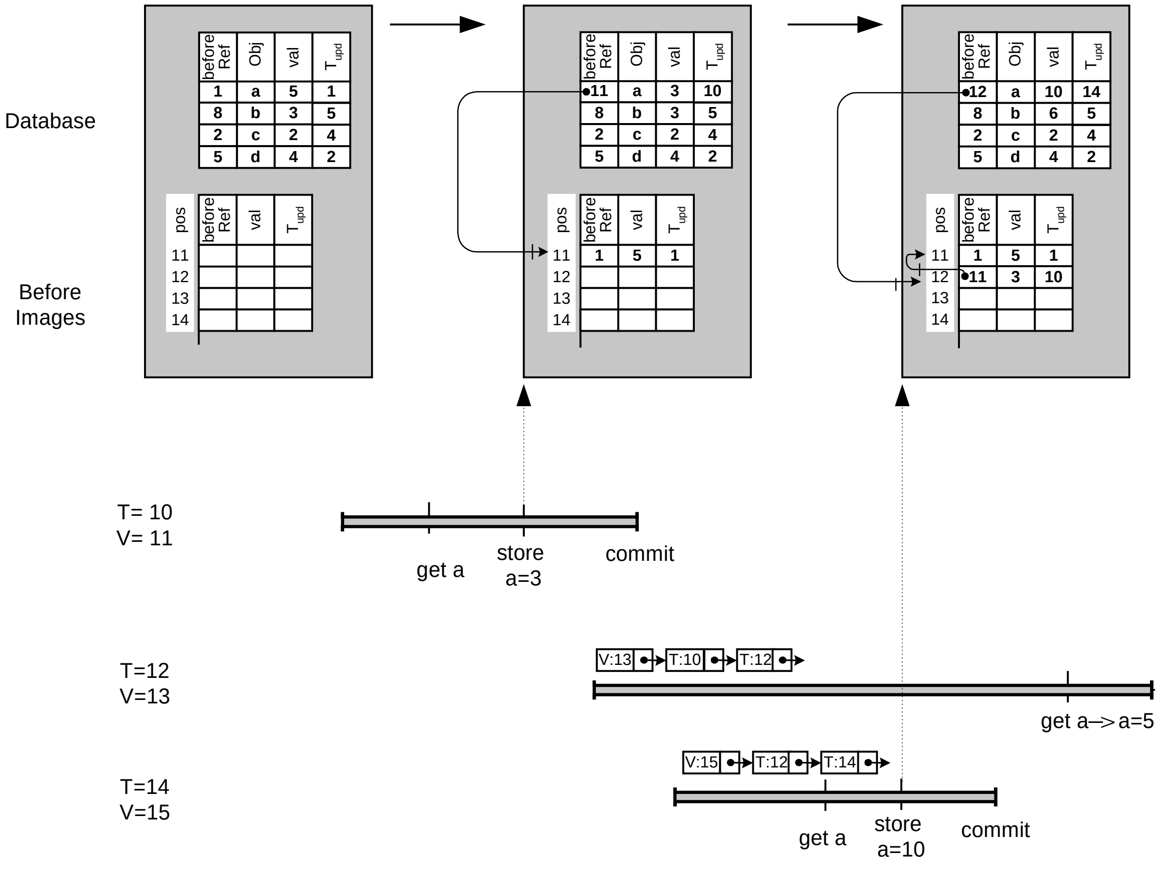
Check Openlist

Secondly, the transaction manager checks if the current object value is committed from the point of the consistent view V=13. It is checked if the transaction which last modified object a (T_{upd}) is in the openlist which belongs to the consistent view. In this example, T_{upd}=10 is in the openlist of the consistent view V=13 and therefore the current object value is not valid. It is necessary to read the before image of the object.

Read Before Image

The object frame includes the reference to its before image (beforeRef). The transaction manager reads the before image to which beforeRef points and checks again if the value is committed from the point of the consistent view. The value of the before image is a=5. Transaction T_{upd}=1 is not in the openlist. Therefore value a=5 is valid.

Figure 9 shows another example. In comparison with Figure 8 this example is more complex. When transaction T=12 reads object *a*, it reads the before image of the before image from the current object *a* with value a=5. This is because the value a=10 and its before image with a=3 were not committed at the time the consistent view V=13 of transaction T=12 was created.



beforeRef	Obj	val	T_{upd}
1	a	5	1
8	b	3	5
2	c	2	4
5	d	4	2

pos	beforeRef	val	T_{upd}
11			
12			
13			
14			

beforeRef	Obj	val	T_{upd}
11	a	3	10
8	b	3	5
2	c	2	4
5	d	4	2

pos	beforeRef	val	T_{upd}
11	1	5	1
12			
13			
14			

beforeRef	Obj	val	T_{upd}
12	a	10	14
8	b	6	5
2	c	2	4
5	d	4	2

pos	beforeRef	val	T_{upd}
11	1	5	1
12	11	3	10
13			
14			

Figure 9 Consistent Read (example 2)

The transaction manager reads the consistent object in the following way:

The transaction T_{upd}=14 of the current object frame is not less than the consistent view V=13. This means the transaction which last modified object *a* was started after the consistent view. Therefore a=10 is invalid from the point of the consistent view.

The before image of object *a* is found at position 12 in the before image storage. The transaction T_{upd}=10, which wrote object value a=5 of the before image, is in the openlist. Therefore, this before image is not valid. It is necessary to read the before image of the before image which is found at position 11. The value a=5 of this before image is the consistent object from the point of the consistent view V=13 because transaction T_{upd}=1 started earlier than the consistent view V=13 and T_{upd}=1 is not in the openlist.

Figure 10 shows an activity diagram of the consistent read. As mentioned above, the transaction manager performs two kinds of checks.

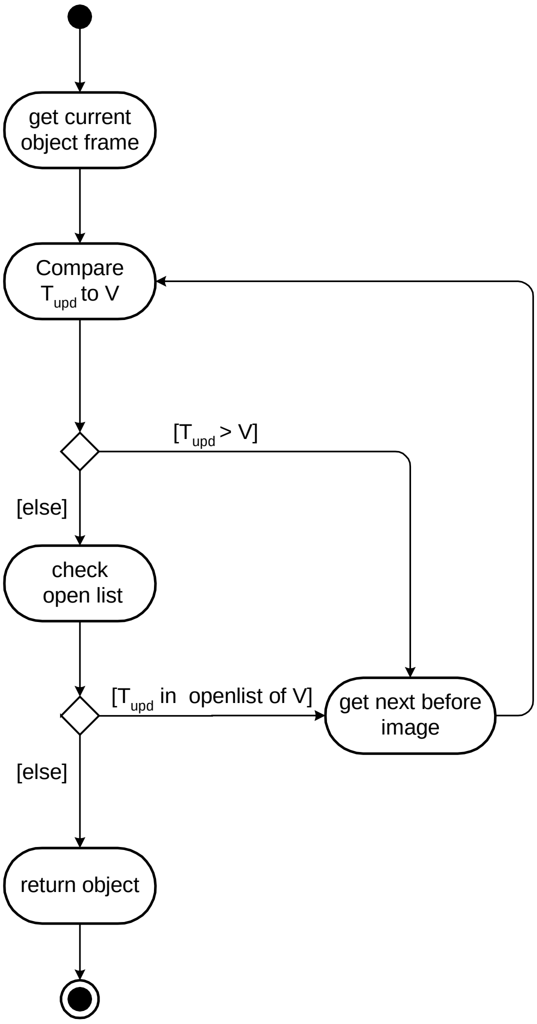


Figure 10 Consistent Read

Check T_{upd}

First, the check if the object was changed by a transaction which started after the consistent view is performed. This is done by comparing T_{upd} of the object frame respectively the before image with the consistent view ID V of the current consistent view. If T_{upd}>V, the transaction started later and the check is repeated with the before image because the current object value does not fit the consistent view. Otherwise, the second check is performed.

Check Openlist

It is checked if the current object value is committed from the point of the consistent view. This check is performed by verifying if the transaction T_{upd} is in the openlist. If so, the current object value is not valid from the point of the consistent view and the two checks are repeated with the next before image. Otherwise, the consistent object is returned.

3.2.2 Lock Management

The lock information is stored in the object frame. The object frame does not only include the beforeRef, the object ID, the value and the ID of the transaction which wrote the value, but also the information on which transaction holds a lock on the object and at which index in the transaction list the transaction ID is to be found.

Furthermore, the transaction administration data includes a transaction list, in which all transactions currently running are listed. It is implemented as an array.

A lock entry in an object frame is only valid if the transaction which holds the lock is still running. This means that the transaction list still includes the same transaction ID at the index which is part of the lock entry.

In the Introduction it was already mentioned that a transaction can only get a lock if two preconditions are fulfilled. First, no valid lock should be part of the object frame. Second, the object has to fit to the current consistent view of the transaction. This means that the object should not be changed from the start of the consistent view on.

Figure 11 shows an example of locking. In comparison to Figure 8 and Figure 9, the structure of the object frame is shown in more detail because it shows the lock information (lock). Furthermore, the transaction list is added. The transaction list includes the open transactions (T_{open})

In the following sections the three transactions T=10, T=12 and T=14 are examined.

Transaction T=10

The lock on object *a* of transaction T=10 is granted, because there is no lock entry in the object frame of object *a*. After the lock is granted, the object frame includes the lock information consisting of the index in the transaction list and the transaction ID T=10 of the locking transaction.

Transaction T=12

The first lock request of transaction T=12 is rejected, because transaction T=10 holds a lock on object *a*. After reading the lock information from the object frame, the transaction manager checks the transaction list in order to verify if the transaction is still running. In this example, the transaction ID in the transaction list for index 1 (T_{open}=10) is still the same as the transaction ID contained in the lock information of the object frame. Therefore the lock is valid.

The second lock request of transaction T=12 is rejected, because the current object *a* does not match object *a* from the point of the consistent view of transaction T=12. The lock information in the object frame is not valid, because transaction T=10 is not in the transaction list for index 1 anymore and therefore already closed. Thus, object *a* is not locked.

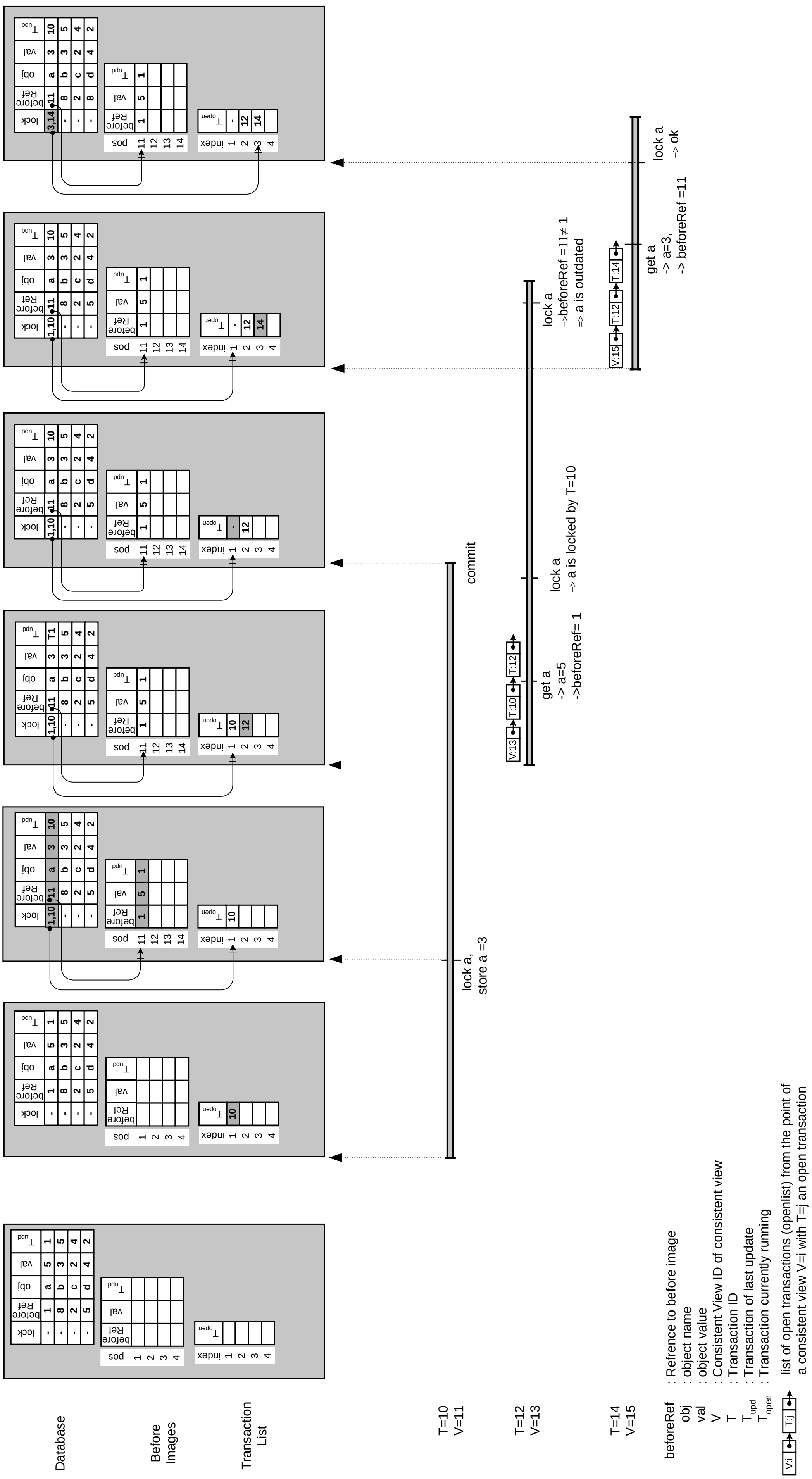

Database
Before Images
Transaction List
T=10 V=11
T=12 V=13
T=14 V=15
lock
beforeRef
obj
val
T_upd
pos
T_open
index
lock a, store a =3
commit
get a -> a=5 ->beforeRef= 1
lock a -> a is locked by T=10
lock a ->beforeRef =11≠ 1 => a is outdated
get a -> a=3, -> beforeRef =11
lock a -> ok
V:13 T:10 T:12
V:15 T:12 T:14
beforeRef : Refrence to before image
obj : object name
val : object value
V : Consistent View ID of consistent view
T : Transaction ID
T_upd : Transaction of last update
T_open : Transaction currently running
V:i T:j list of open transactions (openlist) from the point of a consistent view V=i with T=j an open transaction

Figure 11 Example for Lock Handling

The transaction manager now checks if the beforeRef=1 of object *a*, which was read by transaction T=12, is the same as the current beforeRef=11 of object *a*. This is not the case. Therefore, the transaction manager notices that object *a* has been changed since the consistent view and the lock request is rejected.

Transaction T=14

When transaction T=14 requests the lock on object *a*, the lock entry is not valid and the beforeRef=11 of the consistent object *a* is the same as the current beforeRef=11. Therefore object *a* is currently not locked and has not been changed since creating the consistent view. The transaction manager accepts the request.

Figure 12 shows the activity of the transaction manager handling a lock request.

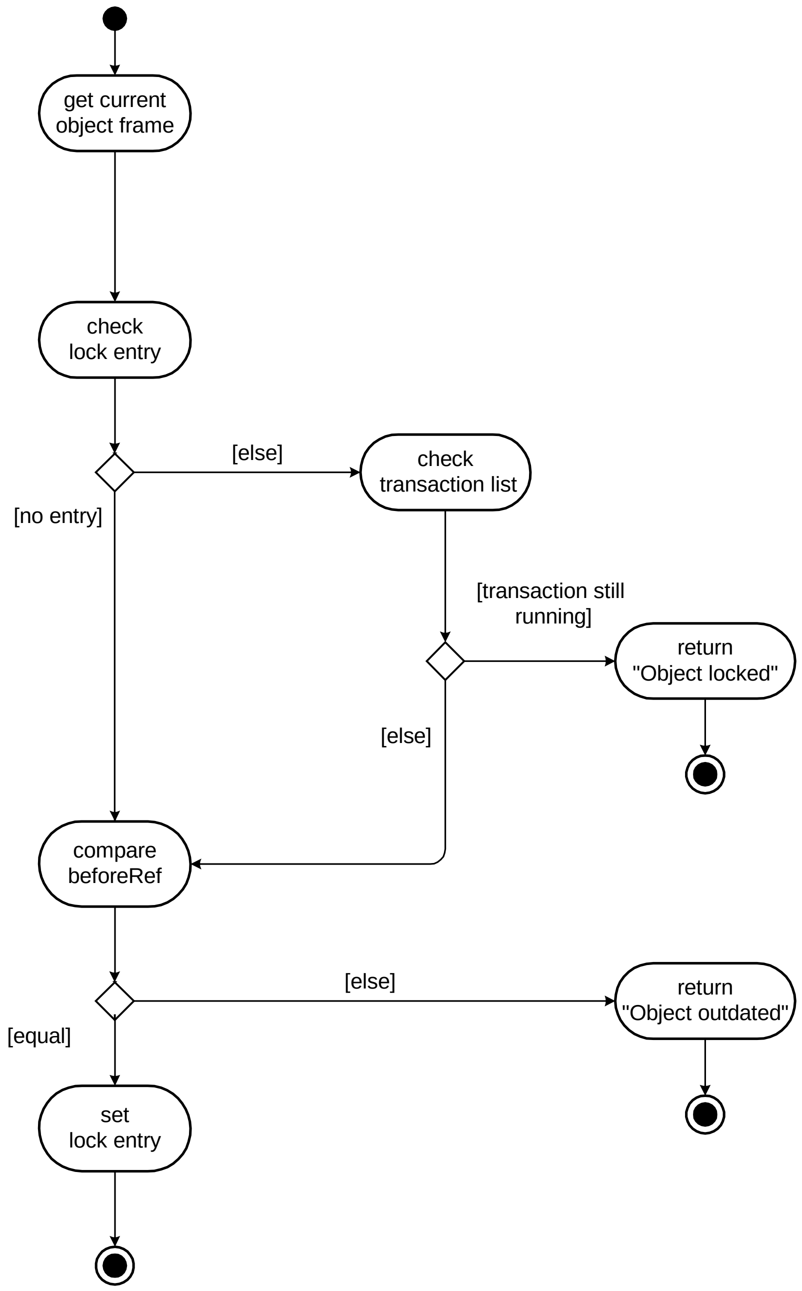


Figure 12 Handling a Lock Request

The transaction manager retrieves the current object frame from the data access manager and checks the lock entry. If no lock entry exists, it directly verifies if the object has changed since creating the consistent view[21]. Otherwise, first the transaction manager checks the transaction list in order to get the information if the lock entry is valid.

If the transaction is still running, the lock entry is valid and the request is rejected with the exception “Object locked”.[22]

If the lock entry is not valid, the check if the consistent object has changed is performed. The beforeRef entry of the consistent object is compared with the current beforeRef. If they are equal, the object has not changed and the lock entry is created, otherwise the request is rejected with the exception “Object outdated”.

3.2.3 Object Frame Structure

As mentioned above, an object is stored in an object frame, which includes additional administrative data. In contrast to the description in 3.2.1, in this section this data is examined in more detail and closer to the actual implementation.

Structure

Figure 13 shows the detailed structure of an object frame. Roughly, it consists of an object header, which includes the administration data of the object, and the object body, which includes the object data and the object key in case of a keyed object.

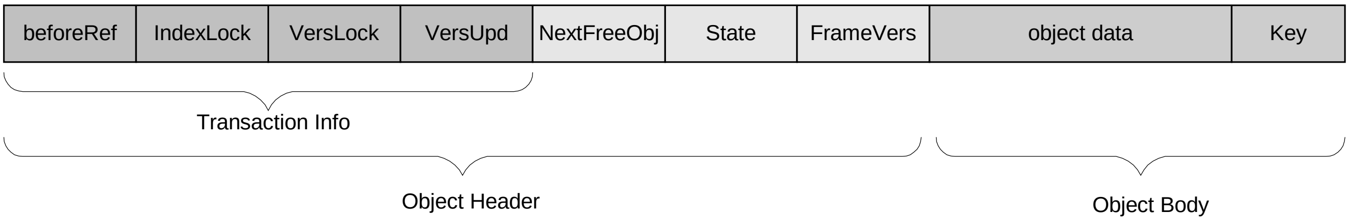


Figure 13 Object Frame Data Structure

Object Header

One part of the object header is the transaction info. The data field beforeRef points to the before image of the object. The fields IndexLock and VersLock form the lock entry of the object. IndexLock determines the index of the transaction which created the lock entry in the transaction list, and Vers-

[21] For the particular lock request when the OMS request manager locks several objects at once, this check is not performed because after successfully locking all requested objects, the OMS request manager starts a new consistent view.

[22] This holds for the lock request being with timout 0. Otherwise, the transaction manager suspends itself and wakes up either when the timeout is reached or when the object is not locked anymore.

lock is the ID of the transaction. A lock is only valid if the transaction ID, determined by VersLock, is still the same in the transaction list at the corresponding index, determined by IndexLock.

The field VersUpd determines the transaction which performed the last update of the corresponding object. It corresponds directly to T_{upd} in the previous examples.

NextFreeObj
State
FrameVers

Furthermore, the fields NextFreeObj, State, and FrameVers are part of the object header. NextFreeObj includes a pointer to the next free object frame in the page. It is only set for free object frames. All free object frames of a page are linked to a free object chain via the NextFreeObj reference. The field State determines if this object frame is free, used, or free after commit. Free after commit means that the object was deleted and that this object frame is free, when the transaction which deleted the object is committed and no other consistent view needs it anymore. The liveCache reuses object frames. This means that an object frame for which the state is free may be used again when creating a new persistent object. If an object frame is reused, the field frameVers is incremented. It determines how often the frame was reused. As mentioned above, an object is referenced by an Object ID (OID). The OID is only valid if it fits the FrameVers entry of the object frame it references.

Object Body

The object body includes the object data and an object key if one exists. The key only exists, if the corresponding object is a keyed object.

3.2.4 Log Entry Structure

In this section the structure of log entries is described in more detail and closer to the implementation. A log entry includes a before or an after image. Object log entries only include before images, they are used for consistent read. SQL log entries may also include after images and are used for database recovery. A log entry consists of administrative data and the log body. The administrative data is split into a general part, which is the same for SQL and object log entries, and a type dependent part. **Figure 14** shows the structure of an object log entry.

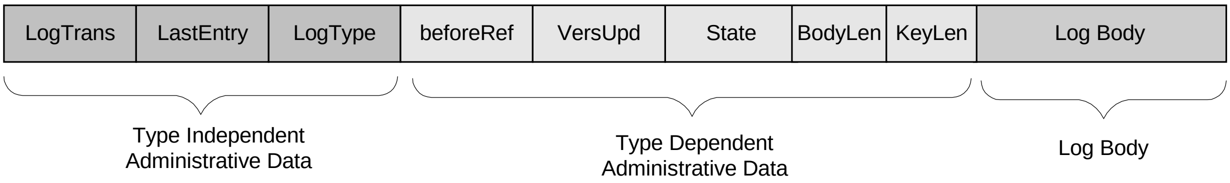


Figure 14 Object Log Entry Structure

General Administrative Data

The general part includes the transaction ID (LogTrans) of the transaction which created the log entry, the reference to the last log entry (LastEntry) created by the transaction and the type information (LogType). The reference to the last log entry is used for rollback to find all before images created by the transaction. The type information specifies which type of request (update, create, delete,...) was the cause for creating the log entry and if it is an object or an SQL log entry.

Dependent Administrative Data

The type dependent part of an object log entry includes the before reference (beforeRef), the transaction ID of the transaction which wrote the object data (VersUpd), and the object state (State). These fields correspond directly to the fields in an object frame described in 3.2.3 and are copied from the object frame when creating the before image. Furthermore the log entry includes the body length and the key length. The body length specifies the length of the log body and the key length the length of the key for a keyed object.

Log Body

The log body is copied from the object body when creating the log entry. If the log body exceeds the log page, it is continued on the follow up page.

3.2.5 Storing, Reading, Creating, Destroying Objects

In this section, the operations on persistent objects from the viewpoint of the transaction manager are described.

Storing Objects

Figure 15 shows a typical example of the sequence of steps when an object modification is stored in the liveCache.

When the OMS request manager sends a “store” request to the OMS Basis, the OID and the pointer to the object in the private OMS cache are sent together with the request.

The OMS basis requests the data access manager to retrieve the pointer to the current object in the page cache. After the data access manager has located the object, it calls back with the object reference as a parameter. Because of the callback, the data access manager further keeps control. This is needed because the data access manager controls internal locks in order to synchronize access to the data structures. For more information about internal locks refer to 3.3.5 and 3.3.6.

Subsequently the OMS basis checks if the current transaction owns a lock on the object. If this check is successful, the object log access manager is requested to write the before image to the object log.

Subsequently, the OMS basis copies the object from the private OMS cache to the page cache and returns to the data access manager, which finally releases the internal locks.

Reading Objects

In case of a read request from the OMS request manager, the OMS basis requests the data access manager to locate the object frame. After the data access manager has located the object, it calls back with the requested object frame as parameter. Subsequently, as described in Figure 10, the OMS basis examines the object frame and, if necessary, determines the consistent persistent object from the log. Finally the call to the OMS basis returns to the data access manager. The data access manager releases internal locks and the consistent object is returned to the OMS request manager.

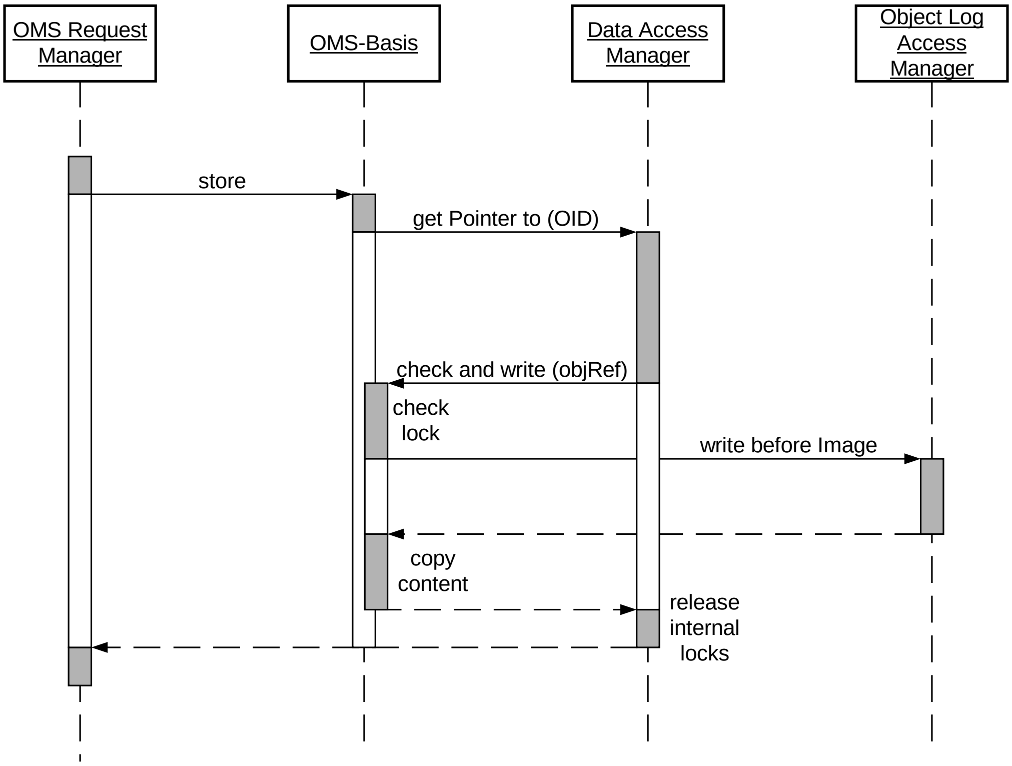


Figure 15 Storing an Object

Creating Objects

When the OMS request manager sends the request to create a new object, it sends the request to the OMS basis together with the class ID and a pointer to the object data in the private OMS cache. The OMS basis requests an empty object frame from the data access manager. The data access manager retrieves an empty object frame in the page cache and calls back with a reference and the OID as parameter. Subsequently, the OMS basis requests the log access manager in order to create a corresponding log entry and copies the object data to the object frame in the page cache. Finally, the call to the OMS basis returns to the data access manager, which releases internal locks, and the OID is returned to the OMS request manager.

Deleting Objects

For deleting an object, the delete request is passed to the data access manager. The process is similar to the modification of an object, except that after successfully checking the lock entry of the object frame, the object is not updated but deleted by the data access agent. In reality, it is not deleted, but the state field in the object frame is set to “free after commit” in order to indicate that the object frame is free once no consistent view needs it anymore. The garbage collector regularly checks if an object frame with state “free after commit” is still needed by a consistent view and sets the state to “free” if possible.

3.2.6 EOT Management

Commit

To commit an OMS transaction, the transaction manager terminates the current transaction and starts a new one. The start of a new transaction also changes the entry in the transaction list. The locks of the previous transaction are released automatically because locks are stored in the object frames and are only valid if the transaction which created the lock is still running.

Rollback

In case of an OMS rollback, firstly the states of the modified objects have to be restored from the log. The log entries created by the transaction are linked backwardly. The transaction is linked to the last log entry it has created. In case of a rollback, the transaction manager requests the log entries in this chain one after the other from the log history access agent. For any log entry the corresponding operation is undone. In case of an object modification, the before image is copied back to the page cache, in case of a delete operation, the state of the object frame is reset to “used” and in case of a “new” object operation, the state of the new object is set to “free” again.

Finally, a new transaction is started and the corresponding entry is modified in the transaction list. Therefore, all locks held by the transaction are released.

3.3 Data Access Manager

3.3.1 Page Cache

The page cache consists of two data parts: Firstly, the cached data, which includes the SQL data, the log pages and the persistent objects and secondly, the cache access administration data, which includes the information which is needed to reference a page in the page cache and the data for the paging algorithm.

Hash function

Figure 16 shows the data structure of the page cache. For addressing a page in the page cache, a hash function is used. The hash function maps a page number to a slot of the hash index. This slot points to the first element of a list of control blocks which belong to pages mapped to the same slot. If the requested page resides in the page cache, then this collision list includes the control block of this page. Every time a page is paged in or out, the hash structure has to be restructured. When a page is paged out, the control block has to be deleted from the corresponding collision list and if a page is paged in, it is added to the collision list of the corresponding slot.

Control Block

The control block contains data for the administration of a page. These are a pointer to the next element of the collision list, the page number (pno), the page state (state), the usage counter (usecnt), the change flag (chgflag), a pointer to a PID queue, a pointer to the next element in the linked LRU list and a pointer to the page in the page cache.

Page State

The state entry is used for the synchronization of the page access. Possible values are state = “io_state”, state = “blocked” or state = “none”. The value “io_state” means that the page is currently read from or written to disk. The value “blocked” means that another task is currently writing this page. The value “none” means that currently the page isn’t written and that no I/O operation is performed on this page.

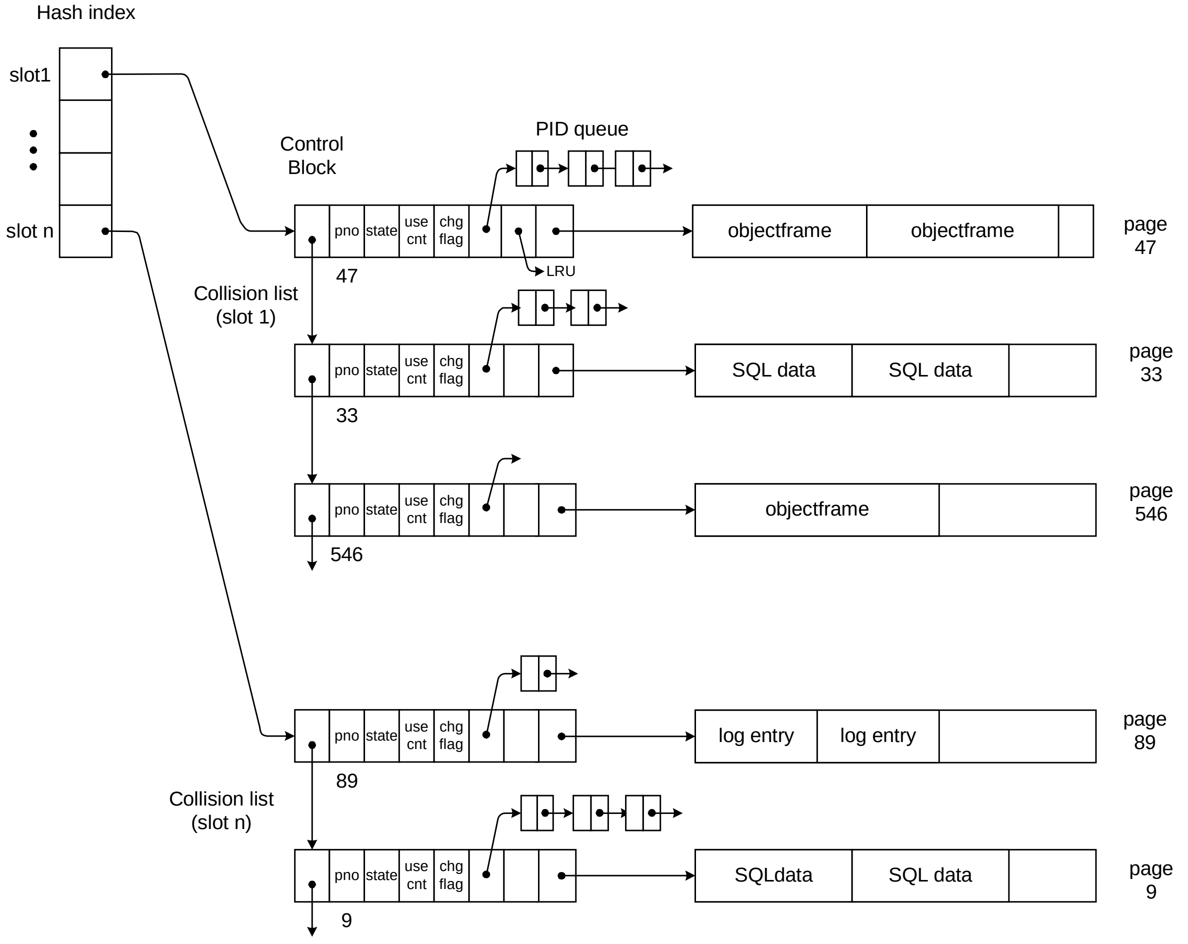


Figure 16 Page Cache Data Structure

Usage Counter

The usage counter counts the number of tasks which are accessing the page concurrently. When a task is writing this page, the value must be 1, because no reader and no other writer may access the page concurrently.

Change Flag

The change flag determines if the page contents differ from the corresponding page stored on the device. If a page is modified in the page cache, this flag is set in order to determine that it has to be written to the device before paging it out.

PID Queue

The PID queue queues the tasks which requested the page, but which have to wait because of synchronization. A task has to wait if the page state has value “io_state” or “blocked”. Furthermore a writer task has to wait if the usage counter is greater than zero. In this case, the task creates an entry in the PID queue. Once the operation has been completed and the page is accessible again, the next task in the PID queue is resumed.

LRU The LRU[23] list, which is only suggested in Figure 16, is needed for the realization of the paging algorithm. It defines an order on the pages in a way, that the pages which where most recently accessed are at the beginning of the list. The elements at the end of the list point to the pages which have not been accessed for a long time. These are the possible candidates for replacement because of paging.

Pages The pages may contain log entries, object frames and SQL data. The pages also include administration data, which is not shown in Figure 16. In case of SQL data, for example, there are also pages which only include information which is needed to find a special SQL record. These are the non leaf nodes in a B* tree and form the so-called B* index. Another example for administrative data which is not shown in Figure 16 are the links between pages which include objects. These pages are linked to page chains. This is discussed in more detail later in section 3.3.2.2 of this report.

Synchronization The access to the hash structure has to be synchronized. When a task searches for a page, no other task can access the hash structure in the meantime. To enable concurrent access to the page cache administration data, it is splitted into more than one hash table (refer Figure A2). The pages are distributed over the hash tables in a way, that allows to determine the corresponding hash table through a “modulo” operation. This improves performance because only the pages handled by the same hash table are not accessible concurrently. Figure 16 does not show this fact.

3.3.2 Data Structure

This section explains the structures in which SQL data, persistent objects and log entries are stored. Furthermore, the structure of the OID which is used to reference an object in the liveCache is described.

3.3.2.1 SQL Data

For each database table there is a B* tree. The nodes of a B* tree are data pages. Only the leave nodes are pages which include the SQL records. The non leaf pages include information to navigate through the B* tree to find a specific SQL record. The details of the structure of a B* tree are not discussed in this report

3.3.2.2 Object Data

OID To reference a single object, the liveCache uses a special object ID (OID). **Figure 17** shows the structure of an OID. It consists of the page number,

[23] Least Recently Used

which determines the page where the corresponding object frame[24] resides, and the page offset, which determines the position of the object frame in the page. Furthermore, it consists of the class ID and the version.

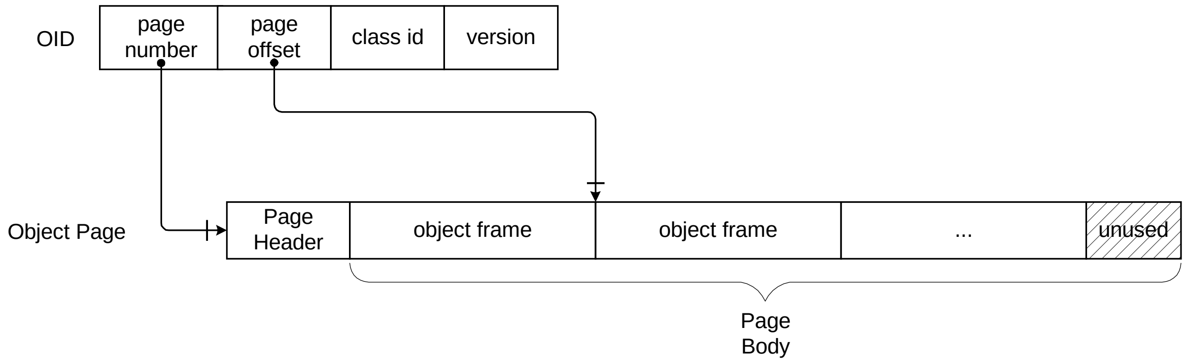


Figure 17 OID and Object Page Structure

Note that the term version in this context has nothing to do with the version described in 2.3. The class ID identifies the class of the object and the version determines how often the object frame was reused after the object stored in this frame was deleted. The version entry must fit the corresponding frameVers entry in the object frame and the class ID must fit the class the referenced object belongs to. This is an additional consistency check. If this check is not performed successfully, an invalid OID is used, which points to an object which does not exists. In case of a different version entry it may be that the object was deleted and the object frame was reused by another persistent object.

Container

Before a persistent object can be stored in the liveCache, the class of the persistent object has to be registered. When a class is registered, the liveCache creates a container for objects of this class. A container is the data structure in which the persistent objects of the corresponding class are stored. In special cases, a class may have more than one container. In this case the OMS request manager specifies the container it wants to access. A container consists of doubly linked, fixed size page chains. The page size is determined during compile time and is between 8 KB and 64 KB. The maximal possible objects size of persistent objects handled in the LC basis is limited by the page size. A page consists of one or more object frames which include the object data. The object frames included in one container always have the same size.

Container Structure

Figure 18 depicts the data structure of a container. The root page of the container includes administrative data and the references to the page chains belonging to the container. The first page of each page chain is named sub root. As opposed to the other object pages, the sub root does not include any object frames. Instead, it includes administrative information about the page

[24] For a detailed description of the object frame structure refer to section 3.2.3

chain, which consists of the fields #pages for the length of the page chain, #free pages for the total amount of empty pages in the chain and the field #free after commit for the amount of pages which include object frames with state “free after commit”. The last part of a sub root page is unused.

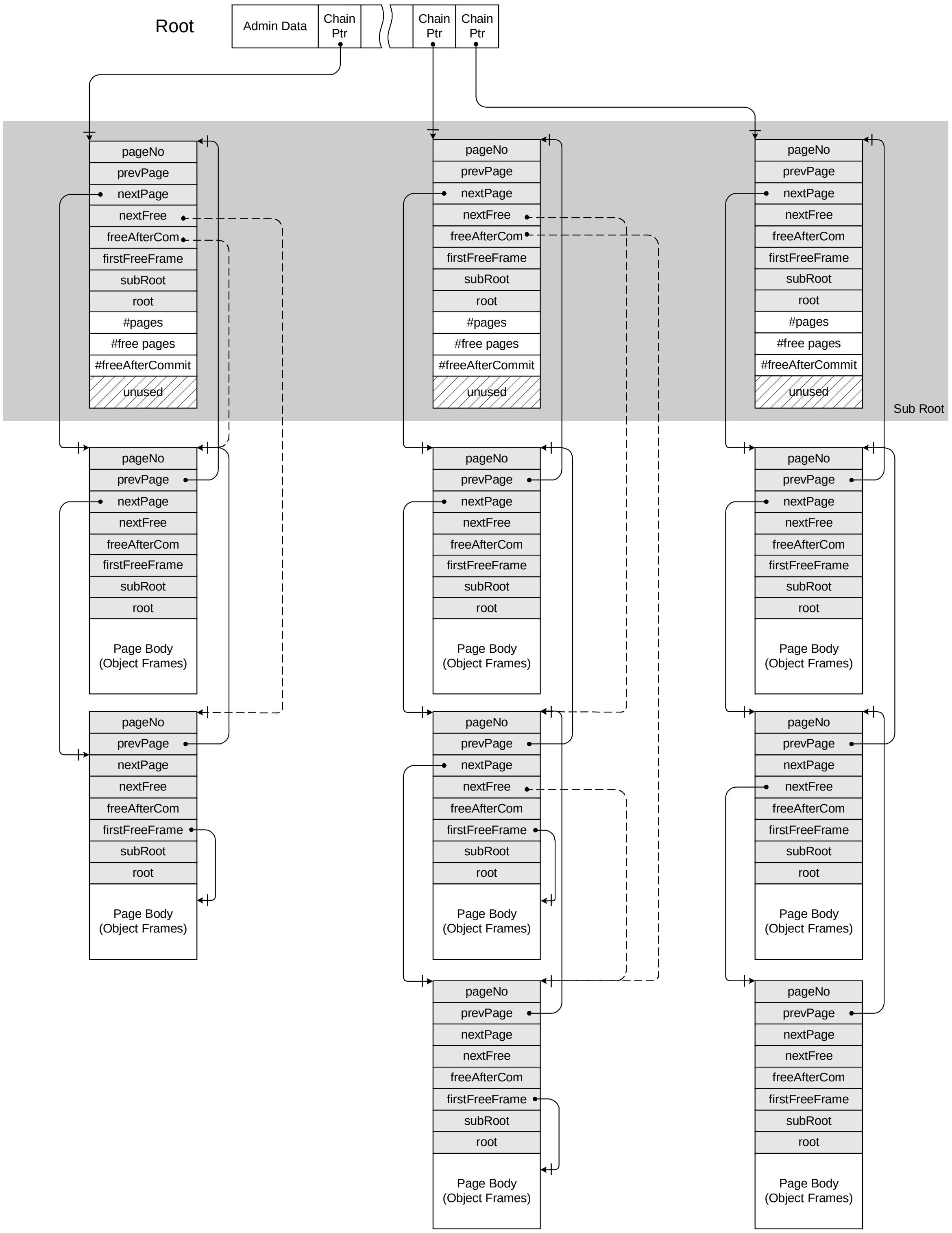


Figure 18 Container Data Structure

Every page consists of the page header (light gray in Figure 18) and the page body, which includes the object frames for pages below the sub root and administration data in case of the sub root. The page header includes additional administrative information.

Page Header

The fields prevPage and nextPage are links to the previous and the next page in the page chain. Additionally, there are links which are needed for the free memory management. The field nextFree points to the next page in the page chain, which includes a free object frame. It is needed by the object access agent in order to find a page with a free object frame when a new object is to be created. The field freeAfterCommit points to the next page in the page chain, which includes an object frame with the state "free after commit". The garbage collector uses this link in order to find object frames with state "free after commit" and to check if they are to be made free in the meantime. The entry firstFreeFrame points to the first free object frame in the page. The object access agent uses this entry to find a free object frame in a page when a new object is created. For the sub root or pages without free object frames the entry firstFreeFrame has value "null".

Furthermore, the page header includes a reference to the sub root of the page chain and a reference to the root of the container.

Page Body

For pages below the sub root the page body is partitioned into fixed size object frames (refer to Figure 17). The last part of the page body may remain unused (dashed area in Figure 17). As mentioned above, for the sub root pages the page body includes additional administrative data.

Multiple Page Chains

Multiple page chains are used in order to improve performance. Chain in and chain out operations on the page chain have to be synchronized. Pages which are completely free are periodically chained out from the page chain. When creating a new object and no free object frame is found, an empty page has to be chained in. During the chain in or chain out operation the chain is not accessible for any other task. Therefore multiple chains improve concurrency, because the other page chains are not affected by such an operation.

Variable Object Length

A special kind of persistent objects are the variable length objects. They can vary in length during their lifetime. As mentioned before, a container only contains fixed size objects. Therefore a special solution is used for objects with variable length. They are stored in a set of containers. Each container of this set includes object frames with a fixed size, which is different for each container. When storing a variable object, the container is selected which fits best. A container fits best when the container preceding in size is too small to store the object. If in case of an update a variable object does not fit in its container anymore, it is deleted in the container and newly created in another container which fits the object size. In contrast to the fixed size persistent objects, variable length objects do not have a container for each object type (class). There is only one set of containers which is used for all variable length objects. Different classes for variable length objects are not supported by the LC basis.

Containers for Variable Objects

The first container of the set of containers for variable length objects includes the smallest objects. It is called the anchor container. If an object can not be stored in the anchor container it is stored in one of the follow up containers and the anchor container contains a frame with all the administrative data of the persistent object and the reference to the object body in the follow up container.

The size[25] of the successor container is a multiple of the current container with a special factor which is the same for all containers. Additionally the size is increased by a value for optimization. To be more precise: $length_k = length_1 * v^{k-1} + c_k$ with $length_k$ the length of object bodies in the k^th container and v the factor. The value c_k is used in order to optimize the frame size so that the whole page is filled with frames with less overhead. The overhead is determined by the page cut and the administrative data in the object header. **Figure 19** shows the amount of page memory which is available for storing the objects, depending on the object size, assuming a page size of 8100 bytes[26] and an object header of 36 bytes. The smaller an object frame, the more object frames fit in the page, which has the side-effect that more storage is used for administrative data. A local maximum means that the page is filled with object frames with no cut. The value c_k is the difference between $length_1 * v^k$ and the next local maximum. The amount n of containers in the set may be calculated with the condition $length_1 * v^{n-1} <= p < length_1 * v^n$ where p is the page size. The parameters v and p are determined before compiling the liveCache kernel, $length_1$ is a start up parameter.

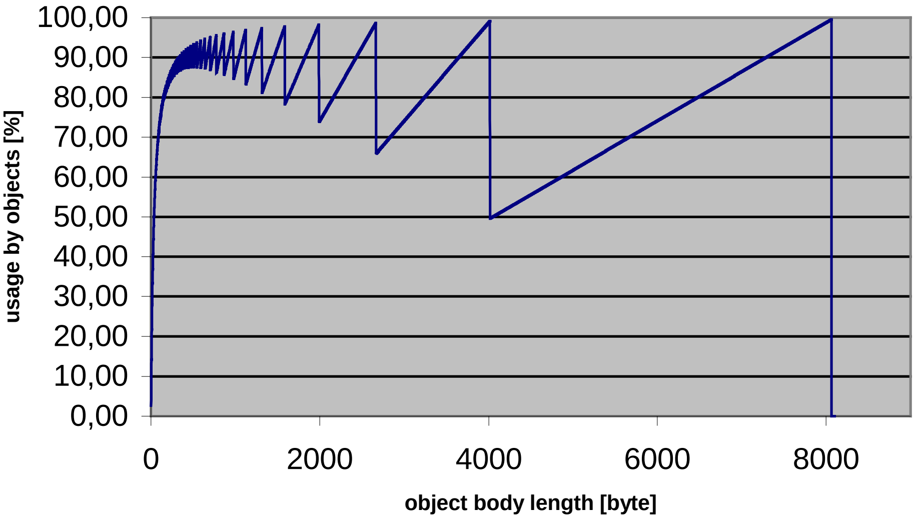


Figure 19 Page Space available for Objects dependent on the Object Body Length

If for example v=2, $length_1 = 60$ and p=8100 then the number of containers is 8 because $60*2^7 = 7680 <= 8100$ and $60*2^8=15360 > 8100$. The length of the 7th container is calculated as follows: $length_7 = 60*2^6 + c_7= 3840 + c_7$. The next maximum is at 4050-36 = 4014. Therefore $length_7 = 4014$.

[25] container size in this context means the size of objects bodies the container may store.

[26] To be precise, the page size is 8192 Bytes. But 92 bytes are used for page administration and therefore not considered here.

Object Frames in the Anchor Container

In contrast to normal containers, the frames in the anchor container include additional administrative data. **Figure 20** shows such an object frame and its relationship to its follow up container. In addition to the object header[27], the object frame of an anchor container includes the length of the variable object. Furthermore, if the object frame is stored in a follow up container, the frame includes a reference to the follow up container as well as the page and the offset of the object frame where the object body is stored.

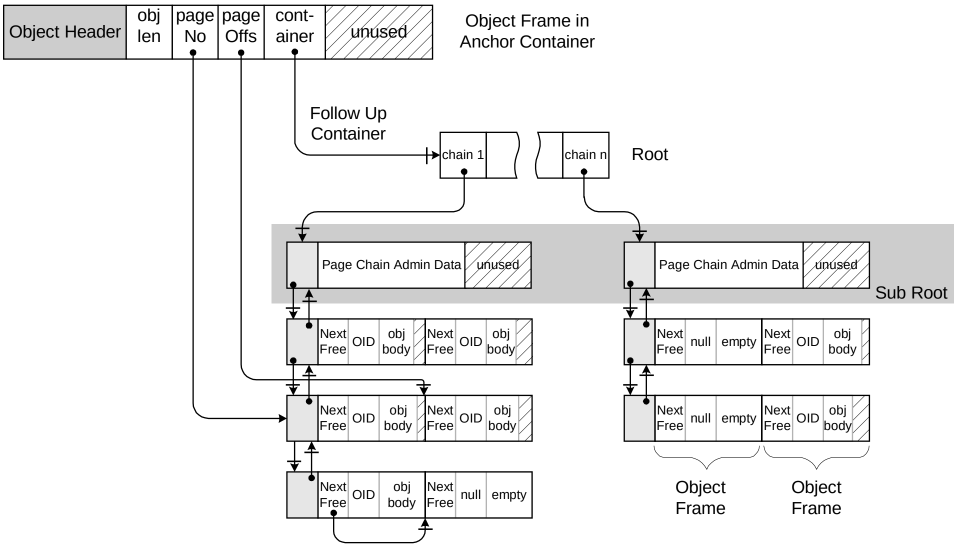


Figure 20 Anchor and Follow Up Container

This information about the variable object will always reside in the same frame until it is deleted, even if the object body changes the follow up container due to a resize operation. The OID of the variable length object is determined by its position (page and offset) in the anchor container and therefore is fixed, even if the object body changes the container.

If the object body does not fit into the anchor container (as depicted in Figure 20), the memory in the object frame of the anchor container remains unused (dashed area in the object frame in Figure 20).

Object Frames in the Follow Up Container

The object frames of the follow up containers differ from the normal object frames in that they do not include the object header, because the header is stored in the anchor container. The only administrative data stored in this kind of object frame is the nextFree pointer, which points to the next free object frame in the page, and the OID, which points to the object frame in the anchor container. The OID is used for an additional consistency check and determines whether the frame is being free or used.

The dashed areas in the follow up container show the memory in the object frames which is unused due to the fact that generally an object body of a variable length object does not fit exactly into the object frame.

[27] refer to 3.2.3

3.3.2.3 Log Structure

This section describes the object log structure. Log entries are sequentially stored on special log pages. For the description of the log entry structure refer to 3.2.4.

Log Structure

The object log is organized as a ring buffer. It consists of a continuous sequence of pages. The log entries are sequentially created and referenced by a log entry ID which includes a page identifier and the page offset. To reference a special log entry, the page number is calculated with the page identifier by a modulo operation over the amount of log pages. Note that log entry IDs can be compared according to their relationship in time.

A log entry may exceed a log page. When it does, it is continued on the follow up page.

3.3.3 Data Access Agent

The data access agent (refer to Figure A2) handles the requests to access the liveCache data. It is divided into the SQL access agent, the object access agent and the log history access agent. At this level, the structure of the different data pages is known. The data access agent requests pages from the page request manager and extracts the requested data depending on the request type (SQL access, log history access, object access).

3.3.3.1 SQL Access Agent

SQL Access Agent

The SQL access agent handles the requests to access SQL records. The retrieval of a record from the view of the SQL access agent is processed along the following steps:

BD Lock Request

Firstly, the SQL access agent requests a BD lock from the BD lock manager. This is a read lock on the whole B* tree (r_lock_tree). This is necessary, because the B* tree must not be changed while a task is searching a record. If the task can not get the corresponding BD lock, it is suspended until the lock may be granted.

B* Tree Navigation

After getting the lock, the root page of the B* tree is requested from the page request manager. If the page is not a leaf, it includes information about which page in the next layer of the B* tree leads to the requested record. This page is requested from the page request manager. The procedure is repeated until the page is a leaf. The leaf page includes the requested record.

BD Lock Request

With a read request, the read lock on the whole B* tree is changed into a read lock on the leaf (r_lock_leaf). For an update request it is changed to a write lock on the leaf (w_lock_leaf). If the B* tree needs to be restructured, the lock is changed to a write lock on the tree (w_lock_tree).

Subsequently, the SQL access agent searches the corresponding record in the page and calls back to the transaction manager, which reads or modifies the record. After the transaction manager has returned, the BD locks are released.

3.3.3.2 Object Access Agent

The object access agent is responsible for accessing the persistent objects in the page cache. The persistent objects may be accessed by a key or directly by specifying the OID.

Keyed Object

When accessing a keyed object, the object access agent has to resolve the key to the OID first. The mapping of keys to OIDs is stored in a B* tree[28]. The key of the record is the object key and the non key elements of the record specify the OID. Therefore, firstly, the object access agent passes a request to the SQL access agent in order to read the OID of the requested object, secondly, the object is accessed by its OID as follows.

Accessing by OID

The page with the page number specified in the OID is requested from the page request manager.

The page offset in the OID determines the position of the requested object frame in the data page. After the requested page is retrieved, the object access agent calculates the reference to the object frame and checks the class ID and the version. If the class ID or the version do not fit the object frame, the OID is invalid. A different version implies that the requested object is already deleted and, resulting from the reuse of object frames, another object resides at the position where the OID points to.

Accessing Variable Length Objects

In case of accessing variable length objects, the OID points to the object frame in the anchor container. If the object body does not fit into the anchor container, the object frame includes a reference to the object frame in the follow up container, which includes the object body.

Callback Transaction Manager

If the requested object frame can be retrieved successfully, the object access agent calls back to the transaction manager with the reference to the object body as parameter. The transaction manager accesses the object data. After the call to the transaction manager returns, the page lock manager is requested to reset the usage counter and if necessary the state entry in the control block of the requested page. The state and the usage counter were set by the cache access manager before.

Deleting Objects

For a request to delete a persistent object, the object access agent determines the object frame in the same way. After retrieving the object frame, the state flag in the object header is set to free after commit in order to indicate that the object frame can be reused after the current transaction has committed and no other consistent view needs it anymore. If the object frame is the only one with state “free after commit” in the page, the page is chained into the “free after commit” list of the page chain and the free after commit entry

[28] This data is not accessible like other SQL data, it can only be accessed internally.

in the sub root is incremented[29]. When deleting a variable object, the state flag is set in the corresponding frame of the anchor container. The garbage collector regularly checks if object frames with state “free after commit” are still needed by other consistent views and if possible sets the state to “free”. For variable objects, also the corresponding frame in the follow up container is set to “free”.

Creating Objects

When the object access agent is requested to create a new object, firstly, the page chain where the new object is to be inserted, is selected by a modulo operation over the number of page chains belonging to the corresponding container with the amount of created objects as first argument. Subsequently, the sub root of the page chain is requested from the page request manager. The nextFree entry of the sub root determines the next page in the page chain which includes a free object frame. If such a page exists, it is requested from the page request manager and the free object frame is determined by the firstFreeFrame entry in this page. Furthermore, the state of the object frame is set to “used” and it is chained out from the list of free object frames of this page. If the page does not include another free object frame, the nextFree entry in the sub root is set to the nextFree page entry of the current page. The free object frame is passed to the transaction manager, which writes the object data. On return, the state and the usage counter of the page are reset. For variable objects, an empty object frame in one of the follow up containers is retrieved if the object does not fit into the anchor container.

Requesting new Object Pages

If no page with a free object frame exists, this is indicated by a “null” value of the next free entry in the sub root, then a new page has to be chained into the page chain. For this purpose, a free page is requested from the page request manager and it is chained in between the sub root and the first page after the sub root. Furthermore, the page header and the page body are initialized. Finally, the first free object frame is passed to the transaction manager. Before inserting a new page, the page lock manager is requested to set the state of the sub root and the first page to “blocked” in order to synchronize the operation with the other tasks accessing the pages.

Creating Keyed Objects

Creating new keyed objects is more complicated, because the keys need to be maintained in addition. **Figure 21** shows the activities when creating a new keyed object.

First, it is checked if the specified key is not in use by another object. In this case the keyed object is created. If the key is used, the object which uses the key is retrieved and its state is checked. If the state has value “used”, the exception “duplicate key” is raised. Otherwise, the object was deleted and it is checked if the delete operation is already committed. This check is performed by trying to lock the object. If the lock is granted, the delete operation is committed and therefore the deleted object is stored as the before image and the new keyed object is created. Storing the deleted object as a before image ensures that it is further accessible by consistent views. If the delete operation is not committed yet, a rollback would result in state “used” of the object and therefore the exception “duplicate key” is raised.

[29] Otherwise this would have happened before

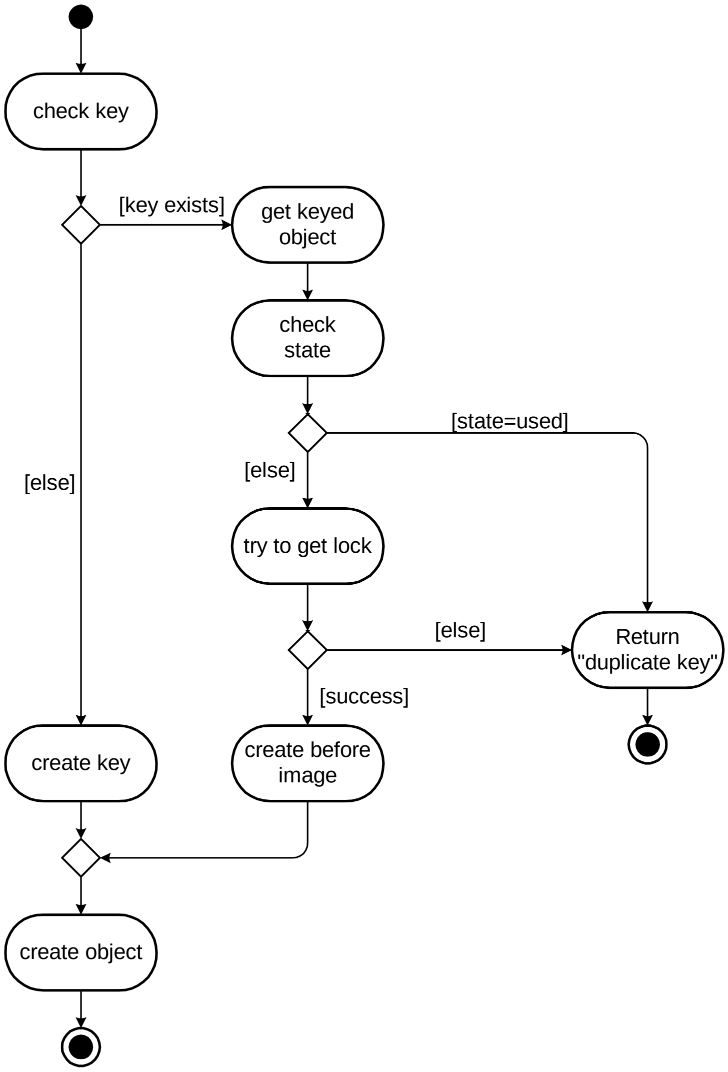


Figure 21 Creating a new Keyed Object

Synchronization

A problem which is not depicted in Figure 21 is the synchronization with the garbage collector when it frees up keyed objects. When preparing object frames of keyed objects for reuse, the garbage collector firstly frees up the object frame and secondly deletes the key. If the object access agent tries to create a keyed object between these two steps with the same key, the key still exists and when checking the persistent object belonging to this key an error occurs because the persistent object has already been deleted by the garbage collector. Therefore the garbage collector has to synchronize with the object access agent for this operation.

Trigger the Garbage Collector

When the object access agent creates or deletes an object, it always accesses the sub root of the page chain in order to check and maintain the administrative data. Therefore it can also be checked if a number of pages in this page chain include object frames with state “free after commit”. If so, the garbage collector is triggered, which checks if the state of object frames may be set to “free” and deletes empty pages from the page chain.

3.3.3.3 Log History Access Agent

The log history access agent has two responsibilities. Firstly it receives requests from the transaction manager in order to read log entries and secondly it retrieves requests from the object log writer in order to write log pages to the page cache. The transaction manager can only place read requests to the log history access agent. When creating new log entries, the transaction manager directly requests the log access manager.

Reading Log Entries

When reading log entries, the log history access agent first checks with the help of the page request and cache access manager if the requested page is already in the page cache. If the log page is not in the page cache, it is checked if the page is still in the log cache or log queue by requesting the log access manager. In this case, the page would be copied to the page cache. In the worst case, the log page is only located on the log device and therefore the page request manager needs to read it to the page cache. After the page is retrieved, the log entry is referenced and a copy is returned to the transaction manager. In case the log entry exceeds the page, the log history access manager finds the follow up page in the same way and builds the copy of the log entry by concatenating the parts.

Writing Log Pages to the Page Cache

In order to improve performance, the object log writer writes object log pages not only to the log device, but also to the page cache. For this the object log writer requests the log history access agent. The log history request agent requests the page access agent to retrieve a free data page in the page cache, if necessary after a paging operation, and copies the log page to the page cache.

3.3.4 Page Access Agent

The page access agent includes the functionality to handle the paging and to reference a page in the page cache. It is divided into the page request manager, the cache access manager and the paging manager. At the level of the page access agent, the internal structure of a data page is not known.

Figure 22 shows the activity of the page access agent and its environment when accessing a page. When the page access agent receives the request from the data access agent, it first searches the page in the page cache. If the page exists, depending on the type of request, the page flags are set in order to synchronize the page access[30]. If this is not possible, the page is not accessible and the task waits until it is accessible again[31]. Finally the LRU list is updated and the page is returned to the data access agent.

If the page is not in the page cache, the paging manager selects a page for paging out and the hash table, which maps the page number to the page in the page cache, is updated. Furthermore, the page state is set to io_state in

[30] For read requests, only the usage counter is incremented and for a write requests, the state is set to "blocked" in addition.

[31] In reality, the task is suspended and sleeps until the page is accessible again

order to indicate that the page is currently being read from disk and therefore not accessible. After this, the LRU list is updated.

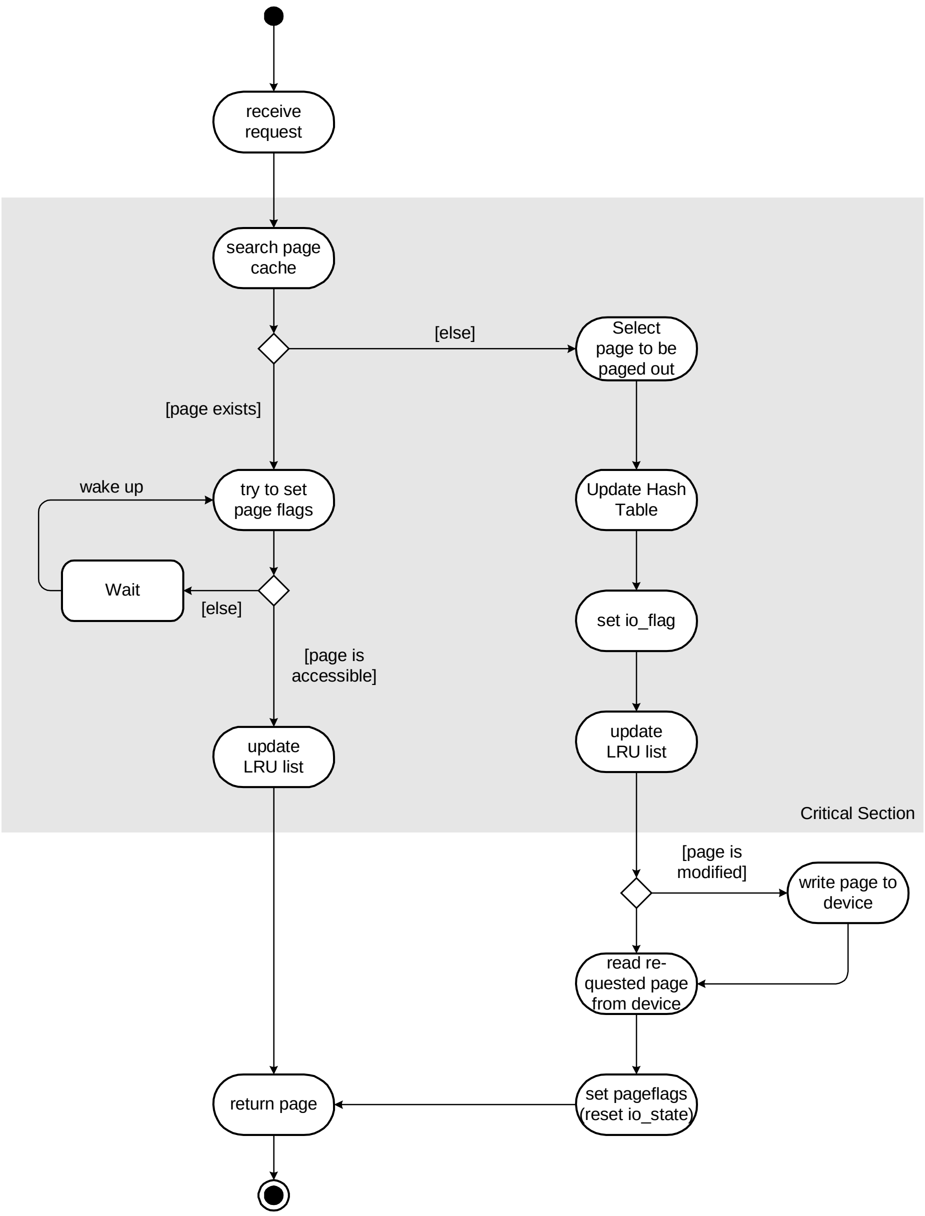


Figure 22 Activity Diagram: Page Access Agent

If the page selected for paging was modified, it is not up to date on the device and therefore is written to disk before the requested page is read into the page cache. Otherwise, the requested page is read to the page cache immediately. Finally, the io_state is reset and the page is returned to the data access agent.

In the following, the components of the page access agent are examined in more detail.

3.3.4.1 Page Request Manager

The page request manager receives its requests from the data access agent. It has to provide a specific page in the page cache.

Firstly, the page request manager passes a request to the cache access manager in order to locate the requested page in the page cache.

Cached Page

If the requested page resides in the cache, the cache access manager returns two references which point to the requested page and to the page control block. The page request manager returns these references to the data access agent.

Page not Cached

I/O

If the requested page does not reside in the cache, the cache access manager returns references to a page and the corresponding control block which may be paged out, and sets the page state to "io_state". If the page is flagged as "modified", it must be written to the data device before reading in the requested page. The I/O operation is executed by the device access agent. After reading in the requested page, the page lock manager is requested to change the state of the page from "io_state" to the state belonging to the corresponding operation. Finally, the references to the requested page are returned to the data access agent.

3.3.4.2 Cache Access Manager

The cache access manager tries to find a requested page in the page cache. If it can not find the page, it requests the paging manager to locate a page which is empty or which may be paged out.

Locating Cached Pages

To locate the page in the page cache, the cache access manager first determines the hash table which administrates the requested page. This happens by a modulo operation on the amount of hash tables.

Accessing the Hash Table

The following operations are synchronized by a critical section. Firstly, the cache access manager uses the hash function to compute the slot of the requested page. The slot is the source for the collision list, which includes the control blocks of the pages mapped to the same slot. Secondly, the cache access manager searches the collision list for the requested page.

Page Exists

If the requested page can be found, the page lock manager is requested to check if the page is currently accessible and to set the usage counter and the state flag if necessary. The page is not accessible if another task is writing this page or if the current task wants to write it while other tasks are reading the page. Another reason may be that the state of the page has value "io_state" due to paging. For SQL pages, the main synchronization is done by the BD lock manager, which protects access to the B* trees. In this case the state flag is necessary for synchronization with the data writer. If the state flag has value "blocked", the data writer can not access the page. If the page is not accessible, the current task is suspended until it may access the page again. Otherwise, the LRU manager is called in order to update the LRU list.

Page not Cached

If the requested page does not reside in the page cache, the paging manager is requested to locate an empty page frame or to select a page for replacement if no empty page frame exists in the page cache. Furthermore, the hash table has to be maintained. The control block of the page which is to be paged out is deleted from the collision list where it is hashed to and the control block of the page which should be paged in is inserted to the corresponding collision list. Furthermore the LRU manager is called to maintain the LRU list and the page lock manager is called to set the state to “io_state” in order to indicate that this page is currently not accessible because of an I/O operation. Finally, a pointer to the control block and the page is returned to the page request manager. At this point the critical section, which protects the access to the page cache, is left.

3.3.4.3 Paging Manager

In case of a page fault the paging manager determines which page has to be paged out or selects an empty page frame if one exists.

It was already mentioned, that the LRU manager maintains the LRU list. The paging manager uses the LRU list to decide which page has to be paged out. It chooses the page which has not been accessed for the longest time and which is currently not in use. This is a page at the end of the LRU list. It passes the corresponding control block to the cache access manager.

3.3.5 BD Lock Manager

BD Locks

As mentioned above, the access to the B* trees and the page chains has to be synchronized. The synchronization of the access to B* trees is implemented by so called BD locks, which are managed by the BD lock manager.

3.3.5.1 BD Lock Data Structure

BD Lock Data

The BD lock manager stores the lock information in the BD lock data, which is divided into the BD lock lists and the administrative data to access the lock lists.

Figure 23 shows the BD lock data structure.

Every B* tree is related to exactly one BD lock list and every lock list is related to exactly one B* tree (refer to right part of Figure 23).

Lock List Elements

The elements of the BD lock list represent the BD lock requests on the corresponding B* tree. An element consists of the lock type (r_lock_tree for a read lock on the tree, w_lock_tree for a write lock on the tree, r_lock_leaf for a read lock on a leaf and w_lock_leaf for a write lock on a leaf), the state of the lock (requested, granted) and additional information, like the task which requested the lock.

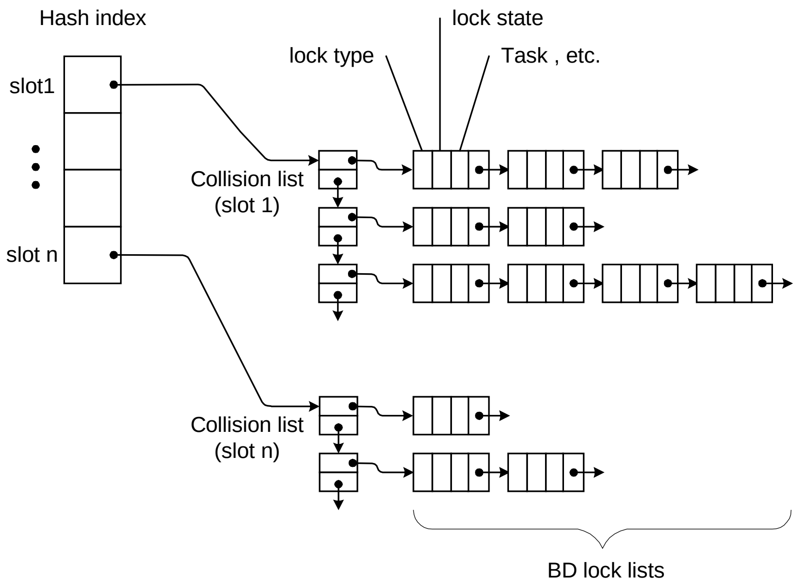


Figure 23 Data structure: BD Lock Data

Hash Index

The relation between a B* tree to its corresponding lock list is organized by a hash index. A hash function maps the ID of the B* tree to a slot of the hash index. A slot points to the first element of the collision list. The elements of the collision list correspond to the B* trees, which are hashed to the same slot. They point to the first element of the corresponding lock list.

Synchronization

The process of searching a lock list by accessing the hash structure has to be synchronized. This means, that only one task at a time can access the hash structure. To improve concurrency, more than one hash structure exists (not shown in Figure 23). The hash structure which maps a specific B* tree to its corresponding lock list can be computed via a modulo operation over the amount of different hash structures.

3.3.5.2 Handling BD Lock Requests

If the SQL data access agent requests a BD lock on a specific B* tree, the BD lock manager first computes the hash structure which manages the requested lock list. The following operations, searching the corresponding lock list, reading the list and creating an entry, is protected by a critical section. The BD lock manager calls the hash function and computes the slot where the requested lock list resides. Furthermore, it searches the collision list for the requested lock list.

Releasing Locks

For a request to release a lock, the BD lock manager searches the lock list for the corresponding lock entry and deletes it. After releasing a lock, it checks if another task is waiting for a lock. This means that it searches for the next lock entry in the lock list with the state “requested”, which could not be granted because of a collision with the lock just released. If another task is waiting for a lock, the BD lock manager resumes this task and sets the state of the corresponding lock entry.

Requesting Locks

A request to get a lock causes the BD lock manager to scan the BD lock list and to check if the requested lock has a collision with one which is already granted. If there is a collision, a corresponding lock entry with state "requested" is created in the lock list and the requesting task is suspended.

If there is no collision, the lock entry is created with state "granted" and the BD lock manager returns immediately.

Furthermore, the BD lock manager provides services to switch from one lock type to another in a single step. If for example an SQL page is found when accessing an SQL record, the read lock on the B* tree is changed to the lock on the leaf in one step.

3.3.6 Page Lock Manager

The page lock manager is responsible for synchronizing page access. It is requested to set or reset the state of a page and to maintain the usage counter.

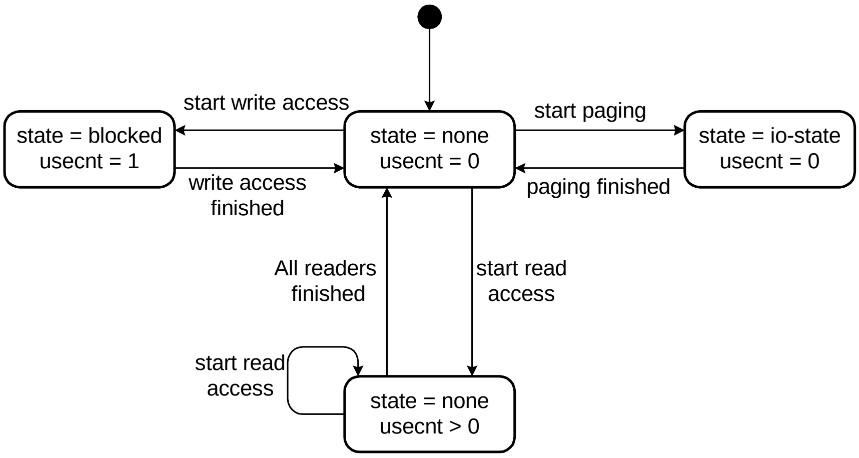


Figure 24 State Transitions of a Data Page

Figure 24 shows the possible states consisting of the page state and the usage counter of a data page and the possible transitions.

The following discusses how the page lock manager handles the different requests.

Read Request

For read requests, the page lock manager is requested to check if no writer and no I/O operation is currently working on the page, and to increment the usage counter if possible. If the state of the page has value "none", the page is accessible for reading and the page lock manager increments the usage counter and returns. During read operations on a page, the state has value "none" and the usage counter is greater than zero. If the state has value "io_state" or "blocked", the page is not accessible for the reader. In this case the page lock manager creates an entry in the PID queue (refer to Figure 16) of the page and the task suspends itself until the page is accessible again.

When a read operation on the page is finished, the page lock manager is requested to decrement the usage counter. If the usage counter is zero, the page lock manager also checks the PID queue in order to verify if a writer is waiting to access the page. If so, the page lock manager resumes the waiting task.

Write Request

For write requests, the page lock manager checks if no other writer, no reader and no I/O operation is currently accessing the page. This means that the state has value “none” and the usage counter is zero. If the page is accessible for the writer, the usage counter is incremented and the state is set to “blocked”. Otherwise, an entry in the PID queue is created and the task is suspended.

After finishing a write operation, the page lock manager is called in order to reset the state to “none” and to reset the usage counter. Furthermore, the page lock manager checks the PID queue and, if necessary, resumes a waiting task.

Paging

For paging, the cache access manager requests the page lock manager to set the state to “io_state”. The paging manager selects a page which is currently not in use. This means that the state has value “none” and the usage counter has value zero. Therefore there is no conflict and the page lock manager sets the state to “io_state”. After the I/O operation finishes, the page lock manager is requested to reset the state and to check the PID queue in order to find out if there is another task which is waiting for the page and to resume it if possible.

3.3.7 LRU Manager

Responsibilities

The LRU manager has two functions. First, it has to restructure the LRU list and secondly it requests the data writer to asynchronously write out changed pages at the end of the LRU list. Writing out these changed pages improves performance because the pages at the end of the LRU list are the typical candidates for displacement. If a page which is to be paged out is not modified, the paging process works much faster because no write operation has to be performed.

LRU List

In 3.3.1 it was already mentioned that the control blocks which point to the data pages are linked in LRU order.

Figure 25 shows the LRU list without focus on the details of a control block. The LRU algorithm used in the liveCache differs from the standard LRU algorithm in that not each page access leads to a replacement in the LRU list.

I/O Area

The LRU list contains the so called I/O area. The I/O area consists of the last part (about 50 %) of the LRU list.

Rechain Area

The very last part (about the last 10 %) of the LRU list is called the rechain area. The pages which are related to the control blocks in this area are the potential candidates for paging. A control block is only replaced in the LRU list if it is part of the rechain area. This improves performance because not for every page access a control block needs not to be rechained in the LRU list.

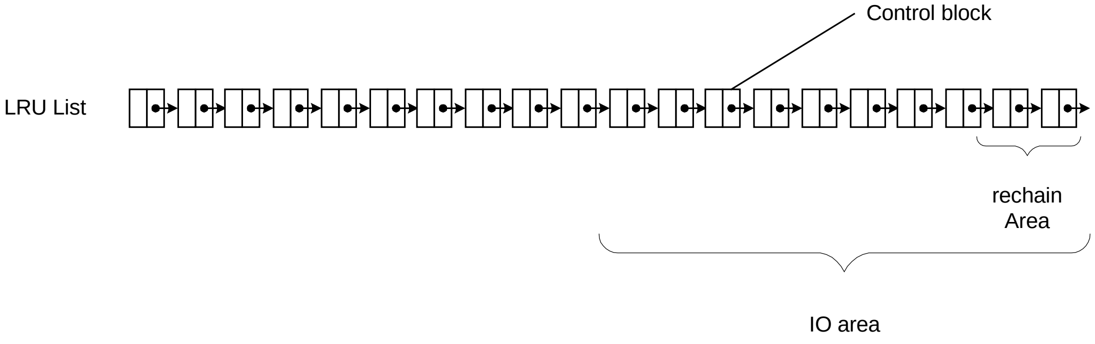


Figure 25 LRU List

Every time the LRU manager is called, it checks if a certain percentage of pages in the I/O area and a certain percentage of pages in the rechain area is flagged as “modified”. In this case the LRU manager requests the data writer to write out these pages asynchronously.

With object pages, only the pages from the rechain area are written to the data device.

3.4 Log Access Manager

The log access manager is responsible for accessing log entries and pages in the log cache. It receives create requests from the transaction manager and read requests from the log history access agent as part of the data access manager. The reason for first handling the read requests by the data access manager is the following: When a log page is full, it is written to the log device and for object log pages additionally to the page cache. Therefore most of the log pages reside in the page cache or the log device. These memory areas are handled by the data access manager. Only if a log page is not in the page cache, the log history access agent tries to get it from the log access manager. The log access manager is divided into the SQL log access manager, which is responsible for handing the SQL log access, and the object log access manager for accessing the object log. In this section, the object log access is examined in more detail. For a description of the object log structure refer to section 3.3.2.3.

Log Cache Administration Data

The last part of the log resides in the log cache. The log access administration data includes the lowest log entry ID (minentry) and the highest log entry ID (maxentry) of the oldest respectively newest entry in the log cache.

Read Request

For a read request the log entry ID of the requested log entry is passed to the log access manager. If the log entry ID is between minentry and maxentry, the log access manager locates the corresponding page in the log cache and returns the page reference to the log history access agent. If the log entry ID is lower than minentry, the log entry does not reside in the logcache anymore. In this case, the log access manager also checks the log queue because it is possible that the entry still resides there. If the log entry ID is higher than maxentry, an error has occurred because this references a log entry which is not yet created.

In case a log entry exceeds the log page, the log history access agent also request the follow up page.

Creating Log Entries

If the transaction manager wants to create a new log entry, it passes a create request to the log access manager. The log access manager passes a pointer to the memory area following the last entry back to the transaction manager and sets maxentry to the new value. Finally, the transaction manager writes the log entry.

3.5 Services

3.5.1 Garbage Collector

The first responsibility of the garbage collector is to check if object frames with state free after commit are still needed by a consistent view and otherwise to set the state to free. Secondly, the garbage collector deletes pages which only include free object frames from a page chain. The garbage collector is triggered by the object access agent when it processes requests to create or delete persistent objects.

Navigation

To check the objects with state free after commit, the garbage collector navigates through the list linked by the free after commit entry of a page, starting with the sub root of the corresponding page chain. For each page in this list, the object frames are checked. If the state of the object frame is free after commit, the garbage collector checks if it is still used. If not, the state is set to free and it is chained into the free object list within the page.

Checking Object Frames

There are three different constellations in which an object frame with state free after commit is still used. **Figure 26** shows these constellations. In Figure 26a, transaction T1 is still open. This means that it is in the current transaction list. Therefore, the delete operation is still not committed. The second case is shown in Figure 26b. The transaction T1, which deleted object *a*, is still in the open list of transaction T2. Therefore T2 needs object *a* for its consistent view. The third case is depicted in Figure 26c. Transaction T2 starts before and ends after transaction T1, which deleted the object *a*. When the garbage collector checks the object frame of object *a*, transaction T1 is already committed, but because T2 started before T1, object *a* is still needed for the consistent view of T2. The LC basis stores the consistent view ID of the oldest consistent view which is still valid. For the constellation in Figure 26c, this oldest consistent view ID is less than the transaction ID which deleted the object.

According to these three constellations, the garbage collector checks three conditions. First, it checks if the transaction which deleted the object is still in the transaction list, secondly, it checks if there is any open list which includes the transaction and thirdly, it checks if the consistent view ID of the oldest consistent view (minview) is before the transaction ID of the transaction which deleted the object.

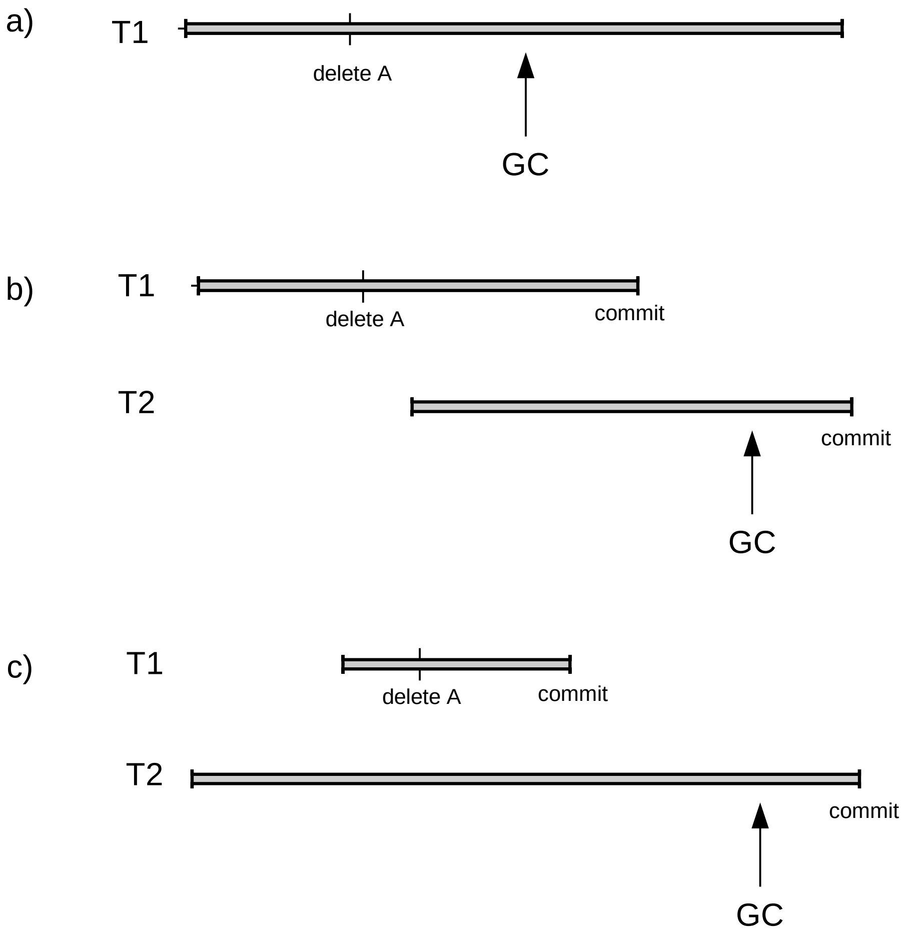


Figure 26 Constellations for used Object Frame in case of Garbage Collection

Freeing up an Object Frame

If none of these conditions applies, the state of the object frame is set to free and the frame is chained into the free object list of the page. In case of a keyed object, additionally the mapping of the key to the OID is deleted and the operation has to be synchronized with the operation of creating a new keyed object. If there is no further object frame with state free after commit, the page is chained out from the free after commit list and the #freeAfterCommit entry in the sub root is decremented. Furthermore, if the object frame is the first one in the page which is set to free, the page is chained into the free list and the #free pages entry is incremented.

If the container is the anchor container for variable objects, the follow up frame, if one exists, additionally is set to free. It is set to “free” by setting the OID entry to null and chaining it into the free object list of the page.

Deleting Empty Pages

In a second phase, the garbage collector deletes the empty pages from the page chain. For this, it scans through the free list of the page chain and deletes any page which includes only empty object frames. If a page is deleted, the state in the control block of the previous page and the next page is set to blocked in order to synchronize the rechain operation. After the rechain operation, these states are reset.

3.5.2 Device Access Agent

If the data access agent wants to read data pages from or write data pages to the data device or if the data writer writes out changed pages, requests are passed to the device access agent.

Reading Pages

The device access agent receives the read requests together with the logical page number and the pointer to the memory area in the page cache.

Converter Table

The converter table includes the mapping of the logical to the physical page number. The device access agent gets the physical page number and reads the page to the page cache.

Writing Pages

Write requests are also received together with the logical page number and a pointer to the page in the page cache. After retrieving the physical page position from the converter table respectively the converter cache, the data is written from the page cache to the data device.

3.5.3 Data Writer

The data writer is responsible for writing changed pages from the page cache to the data device asynchronously. It is triggered by the LRU manager when a certain percentage of pages at the end of the LRU list are flagged as changed. Beyond this the data writer is responsible for writing out pages when processing a checkpoint[32]. In order to synchronize the page access, the data writer requests the page lock manager. If another task is currently writing the page, the data writer is suspended. The data writer requests the device access agent to write the pages to the data device.

3.5.4 Log Writer

The log writer writes log pages to the log device. If a log page is full, the log access manager passes it to the log queue and requests the log writer. An

[32] Checkpoints are not discussed further in this report

SQL log page is inserted into the SQL log queue, an object log page into the object log queue.

The log writer is divided into the SQL log writer and the object log writer.

SQL Log Writer

If the SQL log writer is requested, it checks the SQL log queue and writes the entries to the log file. The writing process is performed asynchronously.

When processing commit, the current, possibly not full, log page is passed to the SQL log queue. Here, the writing process is performed synchronously from the view of the user task. The user task is suspended until the writing is finished.

Object Log Writer

If the object log writer is requested, it checks the object log queue and writes the pages to the log file. In contrast to the SQL log writer, a commit does not cause the object log writer to write out the log page synchronously. The object log writer always writes out log pages asynchronously. Additionally, object log pages are written to the page cache in order to improve performance.

4. Process- and Thread Architecture

This chapter describes the liveCache focusing on the processes and threads assuming a Windows NT environment. The liveCache consists of two processes. These are the XServer and the liveCache process. The XServer connects the liveCache client to the liveCache process. The liveCache process, which is the main process, is responsible for executing the client requests. The process and thread architecture is depicted in **Figure A3**.

4.1 XServer Process

The XServer process is depicted in the upper part of Figure A3. It consists of a single listener thread and connection threads for each connected liveCache client. If a new liveCache client wants to connect to the liveCache, the listener thread receives this request and creates a connection thread, which serves the connection. The communication between the connection thread and the liveCache process is implemented via shared memory and via signal channels.[33] When a connection thread is started, it first allocates shared memory for the communication area and creates its own signal channel. The signal channels are implemented as semaphores. These are needed in order to wake up a thread once a request or response has been sent. Furthermore, the connection thread sends a connect request to the liveCache process and passes the information about its communication area and signal channel with it. This first request is passed to the liveCache process via a mailslot. After the liveCache process has initialized the connection and transmitted its signal channel to the connection thread, further requests may be passed to the liveCache process by writing them to the communication area and sending the wake up signal to the liveCache process.

4.2 LiveCache Process

The liveCache process is more complex than the Xserver process. As mentioned above, it includes all functionalities to process the client requests. The liveCache process consists of the user kernel threads (UKTs) and several service threads. The user kernel threads are the main threads of the liveCache process and are responsible for processing the client requests. A user kernel thread processes several tasks at once. There are e.g. several user tasks. Each of them is dedicated to fulfill the requests of exactly one liveCache client. The user kernel thread manages its own multitasking. The number of user kernel threads running as part of the liveCache process and the number and types of tasks a user kernel thread can execute is configured

[33] When the liveCache client runs on the same host as the liveCache instance, the communication works without the XServer process. In this case, the liveCache client communicates directly with the liveCache process via the shared memory and the signal channels.

at start time and is fixed during runtime. Therefore the maximal possible number of clients is fixed during runtime.

The other service threads include the requestor, which receives and processes the connection requests. The remaining service threads provide services which are processed asynchronously and may be used by the user kernel threads.

4.2.1 User Kernel Thread

Each user kernel thread consists of the task controller and the task handler. The task handler executes the different tasks such as the user task. The task controller controls task execution and handles task scheduling. If for example a user task is waiting for an I/O operation, the task controller resumes another task which is further executed by the user kernel thread.

4.2.1.1 Task Controller

Task Controller

The task controller offers several services which a task may use if it requests memory, performs an I/O operation, requests another task, wants to communicate with the liveCache client, or is waiting for something, e.g. due to synchronization. Furthermore, the task controller receives all client requests as well as requests from other user kernel or service threads and manages the multitasking.

When a task requests the task controller for a service operation which has to be processed synchronously from the view of the task, such as an I/O operation due to paging, the task controller triggers the execution of the request and suspends the task. In case of an I/O operation, the task controller requests the service threads to perform the I/O operation asynchronously from the view of the user kernel thread. While performing the I/O operation, the requesting task is suspended and another task, e.g. a user task, is resumed and may proceed with its work. Later, when the service thread has finished the I/O operation, the task controller resumes the task which is waiting for the I/O.

The task controller consists of the following parts:

Communication Manager

The communication manager is responsible for communicating with the connection threads which are connected to the liveCache clients. A task, e.g. a user task, calls this agent in order to transmit results to or receive requests from the client. The communication manager corresponds to the client communication manager described in section 2.1 as part of the architecture of the client communication layer (refer to Figure A1).

Task Dispatcher

The task dispatcher decides which task may proceed next when a task finishes a request or is waiting for a service. When the dispatcher receives the dispatch request, it checks if requests for other tasks are pending. For this, it checks the external UKT queue (part of the external request queues), the (internal) request queue and the communication area. The external UKT queue includes requests from other user kernel threads or from the service threads. The (internal) request queue queues requests coming from internal tasks. The communication area is checked in order to verify if client requests are pending. If requests are pending, the task dispatcher resumes the next requested task according to the request queues and the communication area by switching to the corresponding task context[34]. Otherwise, the UKT goes to sleep and waits for its wake up signal.

When later on a request arrives, the UKT signal is set and therefore the user kernel thread wakes up, the task dispatcher receives the request and resumes the requested task.

Task Manager

The task manager is responsible for requests which cover task control, such as requests to sleep or to wake up another task as well as to enter or leave critical sections. These requests are received directly from the tasks. The request to wake up another task is queued in the (internal) request queue and in one of the following dispatch operations the corresponding task is resumed by the dispatcher. In case of a sleep request, the task manager calls the dispatcher, which suspends the task and checks for pending requests.

Critical Section

When a task wants to enter a critical section, the task manager checks if no other task is in the critical section at the same time by reading the critical section administration data. If there is no other task, the requesting task may enter the critical section, otherwise the request is stored in the critical section administration data and the dispatcher is called in order to suspend the requesting task and to resume another one if necessary. This means, that the task waits until it may enter the critical section. When a task leaves a critical section, the task manager checks if other tasks are waiting to access it. If so, the next waiting task is selected and a wake up request is passed to the internal request queue when the task belongs to the same UKT[35], otherwise it is passed to the corresponding external request queue.

Service Request Handler

The service request handler provides two kinds of services. The first kind are simple services like read and print which may be processed by the service request handler itself. Other, more complex, services are for example device I/O and backup, which are processed asynchronously from the view of the user kernel thread. These services are not executed by the service request handler itself, instead they are forwarded to the corresponding service thread. If the requesting task has to wait for the requested service[36], the service request handler calls the dispatcher in order to suspend the task and to proceed with another task in the meantime.

After the corresponding service thread has finished, it passes a “finish” request to the dispatcher in order to resume the waiting task. This is done by

[34] The task controller has special priorities to handle the different queues and the communication area. The algorithm for the priority handling is not discussed here.

[35] In certain circumstances it is possible, that due to optimization the task which left the critical section is suspended in order to give the waiting task the chance to continue processing. In general the task will not be suspended and the waiting task is resumed in one of the following dispatch operations, because the wake up request is stored in the internal request queue.

[36] This means from the view of the requesting task, the request is processed synchronously.

inserting the “finish” request into the external request queue of the corresponding user kernel thread and by setting its UKT signal.

Memory Manager

The memory manager is responsible for allocation and release of memory. If a task needs memory or wants to free up memory, it requests the memory manager. In the memory administration data the memory manager stores information about how much memory is in use by which task.

4.2.1.2 Task Handler

User Task Handler

The task handler includes the different handlers for executing the different task types. The main task executed by the task handler is the user task, which processes the client requests. It is worth noting, that the user task handler executes all functions of the architecture described in chapter 2 and 3, except for the basis services (refer to Figure A2) and the client communication manager (refer to Figure A1). As one user kernel thread serves more than one client, multiple user tasks are executed by the user task handler. Therefore, it operates on multiple task contexts. The basis services, described in chapter 3, are executed via the other tasks and the service threads. The service threads are explained later in this chapter.

When a user task finishes a request, it waits for the next request and is suspended by the task dispatcher. In the meantime other tasks are executed.

Server

The server task handler executes the server tasks. A server task is responsible for handling requests which may be processed asynchronously. If for example a user task is requested to create a new index, it requests several server tasks to read the necessary data to the page cache and builds the index in parallel. In general, more than one server task is executed by the server task handler. When a server task finishes a request it is suspended and waits for further requests.

Data Writer

The data writer task handler directly corresponds to the data writer as part of the LC basis services in Figure A2. It executes the request of a user task to write changed pages from the end of the LRU list to the devices (refer to section 3.5.3).

SQL, Object Log Writer

The task handlers for the SQL and the object log writer execute the log writer depicted in Figure A2. These tasks are triggered from a user task when a log page is full. The SQL log writer is additionally triggered when a transaction is committed (refer to section 3.5.4).

Garbage Collector

Another important task is the garbage collector. This task does also correspond directly to the garbage collector described in chapter 3 (refer to Figure A2). The garbage collector task is triggered by the user task in order to delete free object pages and to set the state of deleted object frames to free when they are not needed by any consistent view anymore (refer to section 3.5.1).

Other Tasks

Other Tasks are the utility task, which is only used for monitoring and maintenance, the trace writer, for writing traces, and the backup task for processing the backup. These tasks are executed by the corresponding handlers shown in Figure A3.

4.2.2 Service Threads

The service threads provide services which are executed asynchronously from the view of the user kernel thread, e.g., processing connection requests, reading from or writing to the data devices or processing backup. The service threads communicate with the user kernel threads via the external request queues and signal channels, which are implemented as semaphores.

Requestor Thread

The requestor receives the connection requests from the connection threads. It determines a free user task and requests the corresponding user kernel thread to initialize and start the user task. If no user task is available an error is sent to the connection thread. The maximal possible connections are configured at start time.

Console Thread

The console thread processes requests of the monitoring and maintenance tool “X_Cons” (not shown in Figure A3).

Timer Thread

The timer thread provides a service for controlling timeouts. It is requested by the service request handler if a task requests to sleep for a specified time. After the timeout is reached, the timer thread passes a corresponding “timeout” request to the user kernel thread in order to wake up the requesting task again.

Furthermore the timer thread regularly checks the client connections in order to verify if they are alive.

Clock

The clock thread regularly requests the time from the operating system and writes it to a global data area. Therefore the tasks can read the time from the global data area and do not need to request the operating system.

Async0, Asynci

The async0 and asynci threads serve the asynchronous backup processing. The async0 thread is responsible for opening and closing the backup devices as well as for starting the asynci threads. An asynci thread is started for each backup device and handles the I/O on that device.

Dev0, Devi, Worker

The dev0 and devi threads handle the I/O operations on the data, log and system devices. In analogy to the async0, the dev0 is responsible for opening and closing the devices and for starting the devi threads. A devi thread is started for any data, log and system device and handles the I/O on that device. With a Windows NT operating system, the devi threads are only used if a lot of I/O requests are pending. In normal cases Windows NT executes the I/O operations in the background and notifies a special worker thread if an I/O operation has finished. When the execution of an I/O request has finished, the worker thread or the devi thread informs the user kernel thread by passing a “finish” request into the external UKT queue and by setting the UKT signal. The devi and worker threads correspond to the device access agent described as part of the LC basis architecture (refer to section 3.5.2 and Figure A2).

Dcom0, Dcomi

The dcom0 thread is responsible for starting the dcomi threads. The dcomi threads are responsible for creating the factories, which instantiate the application contexts[37]. These threads execute parts of the application context manager as part of the client communication layer (refer to section 2.5 and Figure A1).

4.2.3 Scenarios

In this section two scenarios are presented. Both focus on how the components of the process and thread architecture are involved. Firstly, the steps of creating a connection to the liveCache are examined. Secondly, it is explained, how a client request is processed.

4.2.3.1 Creating a liveCache Connection

Connection Thread

Figure 27 shows the steps for creating a liveCache connection. The liveCache client connects to the listener thread over the network. The listener thread starts a dedicated connection thread for the liveCache client. Subsequently, the liveCache client is connected to its connection thread.

Sending Request to the Requestor

The connection thread opens the mailslot to the requestor and creates its client semaphore and communication area. As mentioned above, the client semaphore is a wake up channel of the connection thread. Finally, the connection thread wakes up the requestor thread by writing information about its client semaphore and its communication area to the mailslot.

Requesting a User Kernel Thread

The requestor thread reads the information from the mail slot, checks the database state[38] and selects a free user task with its corresponding user kernel thread. Furthermore, it writes a start task request to the UKT queue of the selected user kernel thread and wakes up the task dispatcher by setting the corresponding UKT semaphore.

Task Dispatcher

The task dispatcher reads the request from the UKT queue and requests the task handler to start the user task[39].

User Task

After the user task has initialized, it waits for the connect request which contains additional information. This is information about the client semaphore and the client communication area. Therefore the user task passes a

[37] In case of debugging only an application context proxy is executed in the liveCache context and the application context itself is processed outside the liveCache. Here the dcomi thread additionally handles the calls to the application context via the proxy.

[38] If the database state is “cold”, only connections for administration and monitoring are accepted, user connections are not allowed and the request is rejected.

[39] This only holds if the user kernel thread is idle. In normal cases, the user kernel thread is processing another task when the request arrives. It will check its request queues when the current task finishes or is suspended. It may be that, depending on the request queues, other requests are executed beforehand.

"waitForConnect" request to the communication manager. The communication manager calls the dispatcher in order to check if the request has arrived.

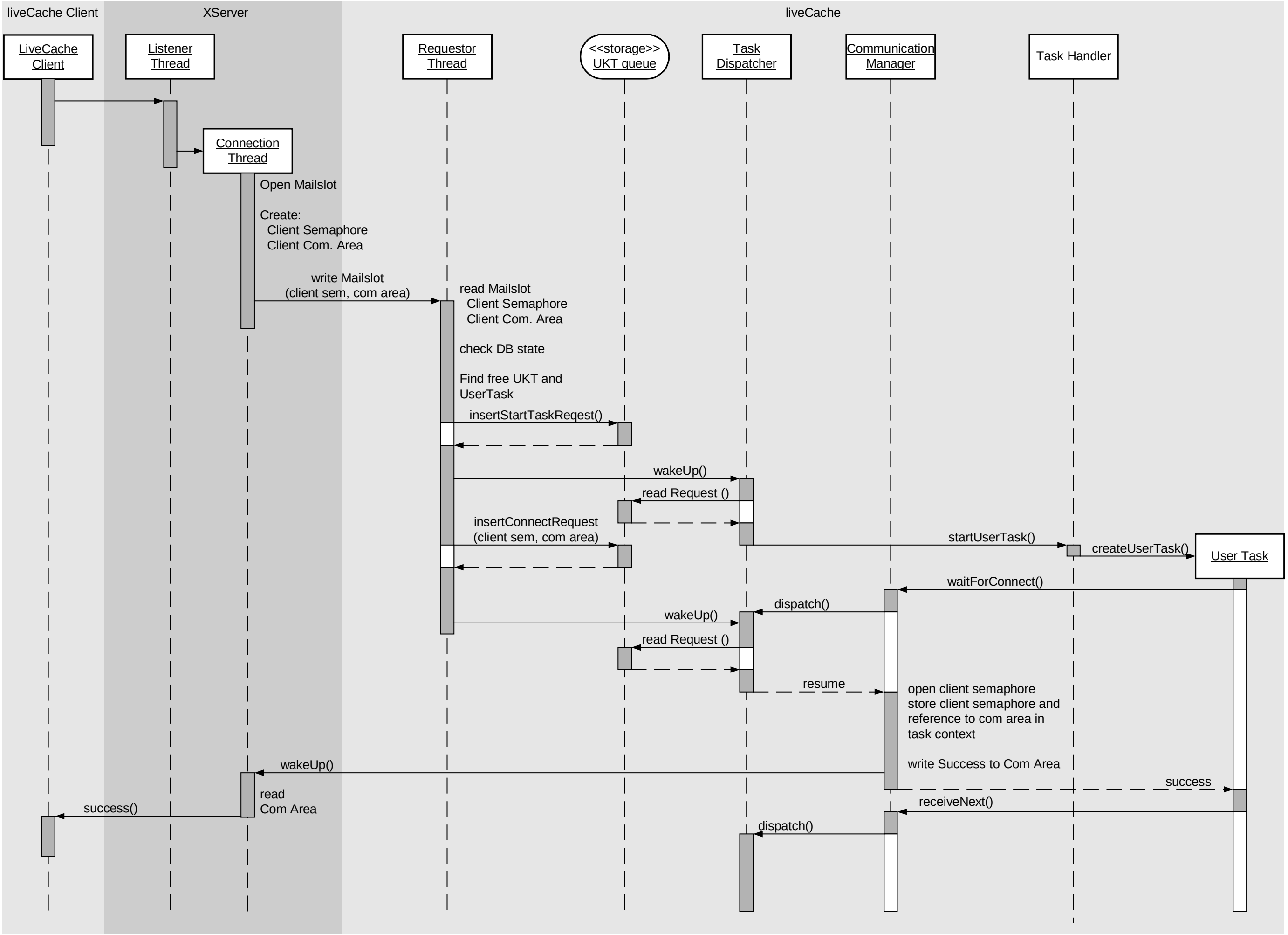


Figure 27 Sequence: Creating a liveCache Connection (best case)

Connect Request

The requestor thread passes this request as a second request to the task dispatcher. It inserts the connection request together with the information about the client semaphore and the client communication area into the UKT queue. In this example, the request has already arrived when the task dispatcher is called by the communication manager. The task dispatcher reads the request and resumes the user task which is waiting in the communication manager.

Communication Manager

The communication manager stores the information about the client semaphore and the client communication area in the task context and writes a success message to the client communication area. Subsequently, a wake up signal is sent to the connection thread by setting its client semaphore. The connection thread reads the success message from its communication area and passes it to the liveCache client. In the meantime, the communication manager returns to the user task, which finally calls the communication manager again in order to wait for the first request. The communication manager calls the task dispatcher, which suspends the task and continues with other requests.

4.2.3.2 Processing a Request

Figure 28 shows the steps of a request processing example. In this simplified sequence, it is assumed that the user kernel thread is idle when the request arrives and that request processing is proceeded without requesting any other services.

Waiting for Request

The upper part of Figure 28 shows the user task requesting the communication manager to wait for the next client request. The communication manager calls the dispatcher in order to suspend the task and proceed with requests for other tasks. The processing of other tasks is not explicitly shown in Figure 28, this is only suggested by the tilde and the dots.

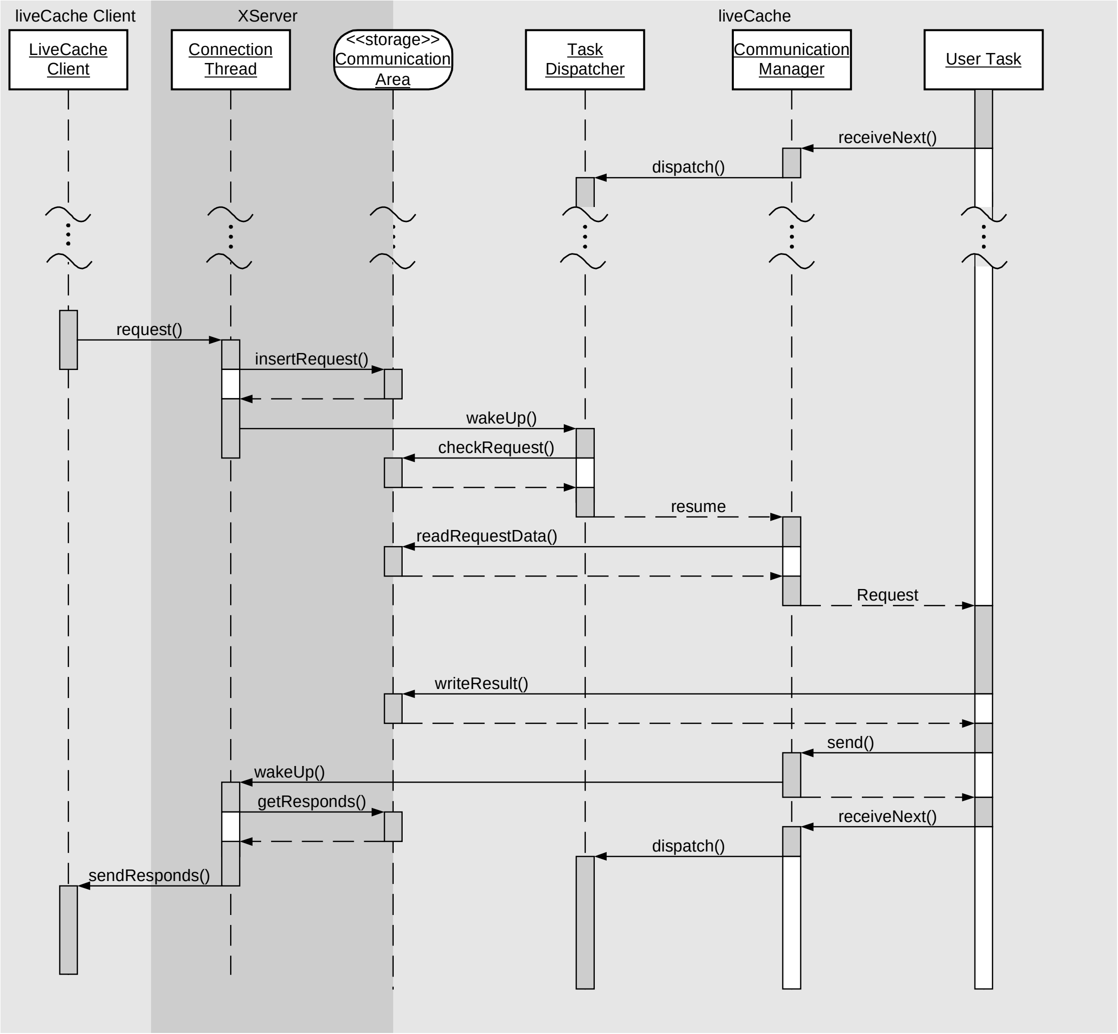


Figure 28 Sequence: Request Processing (best case)

Sending Request

In the lower part of the diagram, the liveCache client passes the request to its connection thread. When the connection thread receives this request, it inserts the request into the communication area. Subsequently, it wakes up the task dispatcher by setting the UKT semaphore.

Task Dispatcher

The task dispatcher reads the communication area in order to check for which user task a request has arrived. The task dispatcher then restores the context of the requested task and returns to the communication manager in which the requested task is waiting.

Processing the Request

The communication manager reads the request data from the communication area and returns the request to the user task. The user task processes the request and writes the results back to the communication area[40].

Reading Results

When finishing the request, the communication manager is requested to send the results to the connection thread. The client semaphore is set and therefore the connection thread wakes up, reads the response data from the communication area and returns it to the liveCache client.

In the meantime the user task requests the communication manager in order to wait for the next request. The communication manager calls the task dispatcher and the user task is suspended again.

[40] To be more precise, the user task writes the results to the communication area while processing rather than by one explicit operation as shown in Figure 28.

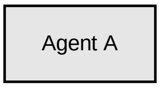
Appendix

A. Modeling Techniques Summary

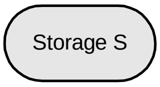
This appendix gives a short overview of the elements in the structure plans used by the Basis Modeling group. For detailed information see the report **Description Method - Modeling Principles and Structure Plans**.

A.1 Block Diagram

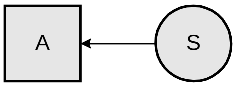
Block diagrams show the structure of a system, i.e. its *components* partitioned in active and passive parts and the *connections* between them. They show the *data flow* in a system.



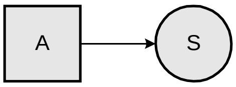
Agent A is an active system component. An agent can be refined with any complex block diagram (i.e. can contain agents and storage places).



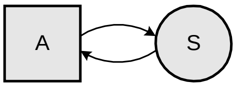
Storage place S is a passive system component. A storage place can be refined with a number of sub-storage places.



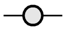
Agent A can read from the storage place S.



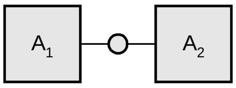
Agent A can write to, but not read from, the storage place S. This can mean a complete or a partial access to the contents of S.



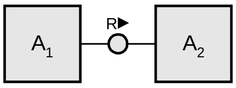
Agent A can write to and read from the storage place S. This can mean a complete or a partial access to the contents of S.



A communication channel is a non-saving, passive component. It can temporarily hold data and events for the duration of a communication.



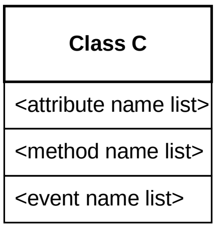
Communication between agent A_1 and agent A_2. The kind and direction of the communication are not specified in more detail.



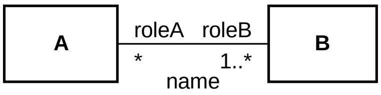
Refinement of the communication channel between two agents. Agent A_1 sends requests (R) to agent A_2 and receives its responses.

A.2 UML Class Diagram

UML class diagrams visualize object classes (often called entity types or object sets) and relationships between them.



Class C. The name is the term of one instance of the class (e.g. the class of all cars is named car). The compartments specifying **attributes** (possibly with type specification) and **methods** (possibly with signature specification) are optional.



Association. Instances of class A are associated with (UML: ‘linked to’) instances of class B. An instance of class A is associated with at least one instance of class B. The number of instances of class A associated with one instance of class B is not specified. In this example A plays roleA and B plays roleB in the association. Name, roles and cardinalities are optional.

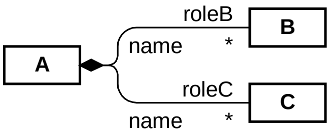
- basic cardinalities:

N	a number n (as variable) or constant (e.g. 5)
*	any number

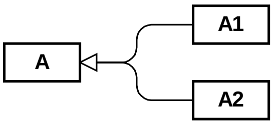
- complex cardinalities:

x..y	Cardinality range, where x and y are basic cardinalities.
a,b,...	Cardinality list, where a,b, etc. are basic cardinalities or cardinality ranges.

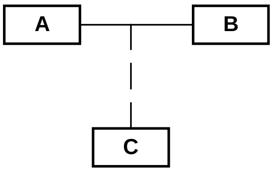
Cardinalities should be expressed as simple as possible, e.g. ‘*’ instead of ‘0..*’.



Composition. It is a special kind of association. Every instance of classes B and C is associated with exactly one instance of class A. If an instance of class A ceases to exist, the associated instances of classes B and C are destroyed (existence dependency). Name, roles and cardinalities are optional.



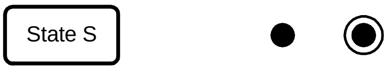
Specialization. The classes A1 and A2 are subclasses of class A.



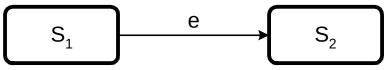
Objectified Association (UML: ‘Association Class’). The elements of the association between the instances of classes A and B are viewed as instances of class C. Thus they can have attributes and methods, can be associated with instances of other classes, and be specialized.

A.3 UML Statechart and Activity Diagrams

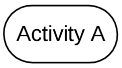
Statechart diagrams show system/object states and events leading to state transitions. An activity diagram is a variation of a statechart diagram, representing all possible sequences of operations in a system or system group. Statechart diagrams consist of states and transitions. The states are represented by state symbols and the transitions are represented by arrows connecting the state symbols. States may also contain subdiagrams by physical containment and tiling. An activity diagram is a special case of a state diagram in which all (or at least most) of the states are activities (action states). A transition leaving an activity is implicitly triggered by completion of the source activity. A statechart diagram shows the state transitions of only one system/object whereas activity diagrams may represent the interaction of several systems/objects.



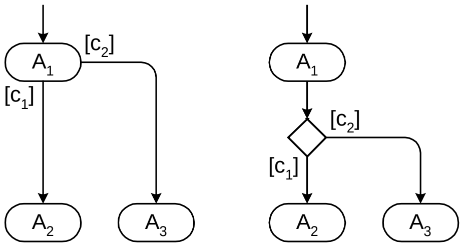
Simple state S, *initial* and *final* (pseudo) state. Inside the rectangle, the name of the state may be shown.



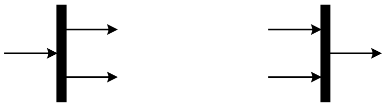
Transition between states S_1 and S_2 triggered by the event e. The text near the arrow denotes the event that triggers the transition.



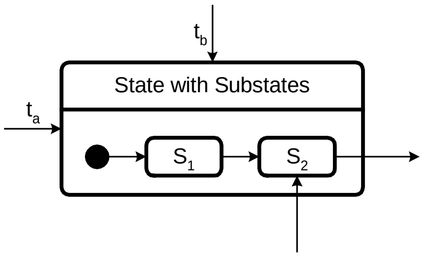
Activity A (action state). The text in the activity describes the actions executed at an arriving transition.



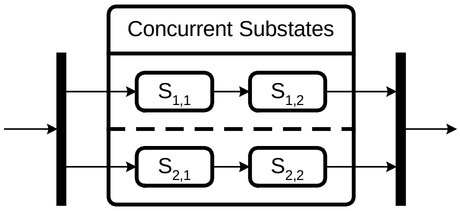
Decision. Multiple output transitions of activity A_1 are labeled with different guard conditions c_1 and c_2. Depending on which condition is fulfilled after execution of A_1, the transition to activity A_2 or activity A_3 is triggered. The figure on the right shows an alternative representation with a *branch* (pseudo) state depicted as a diamond.



Fork/Join. The *fork/join* (pseudo) state is shown as a heavy bar and represents either a splitting of control into concurrent threads (left picture) or a synchronization of concurrent threads (right picture).

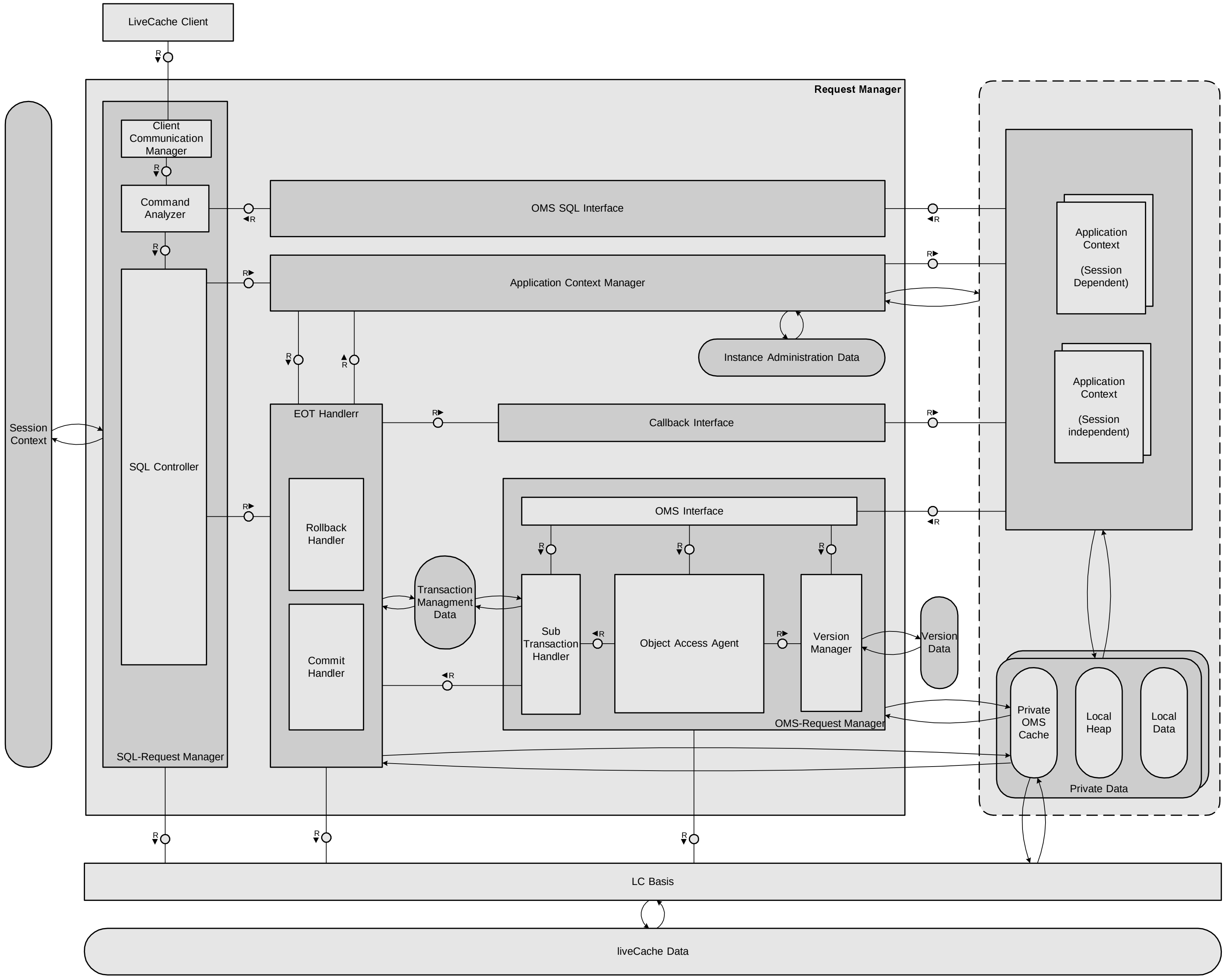


Composite state. A state can be decomposed into mutually exclusive disjoint substates. The transition from the *initial* state inside the composite state to the state S_1 is triggered by any transition to the enclosing composite state, e.g. transition t_a and t_b. Transitions may be drawn directly to or from states enclosed in a composite state.

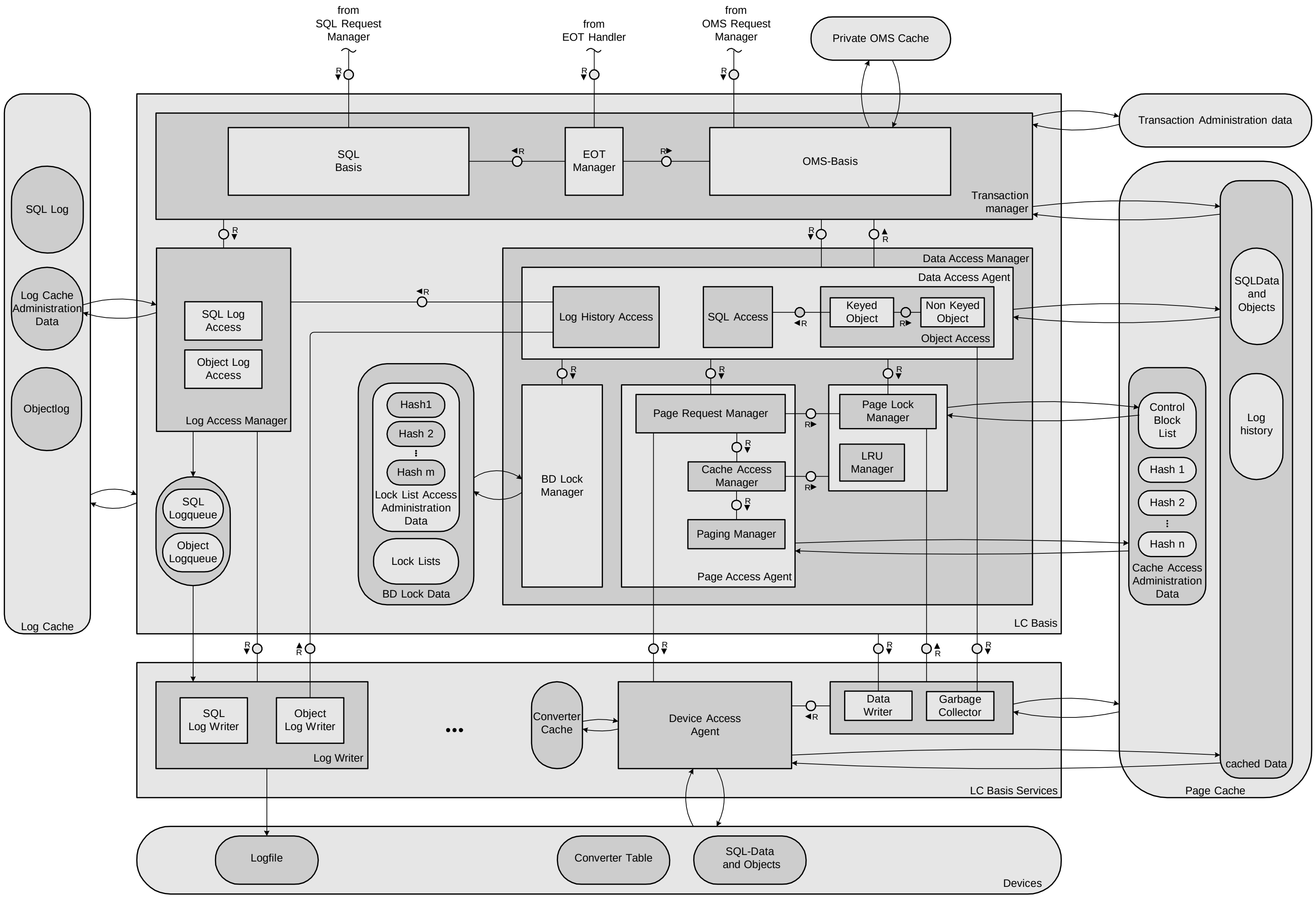


Composite state with concurrent substates. The lower compartment containing the substates is divided into two subregions using a dashed line. Each of those regions is a concurrent substate. The states inside each subregion are again mutually exclusive.

Index



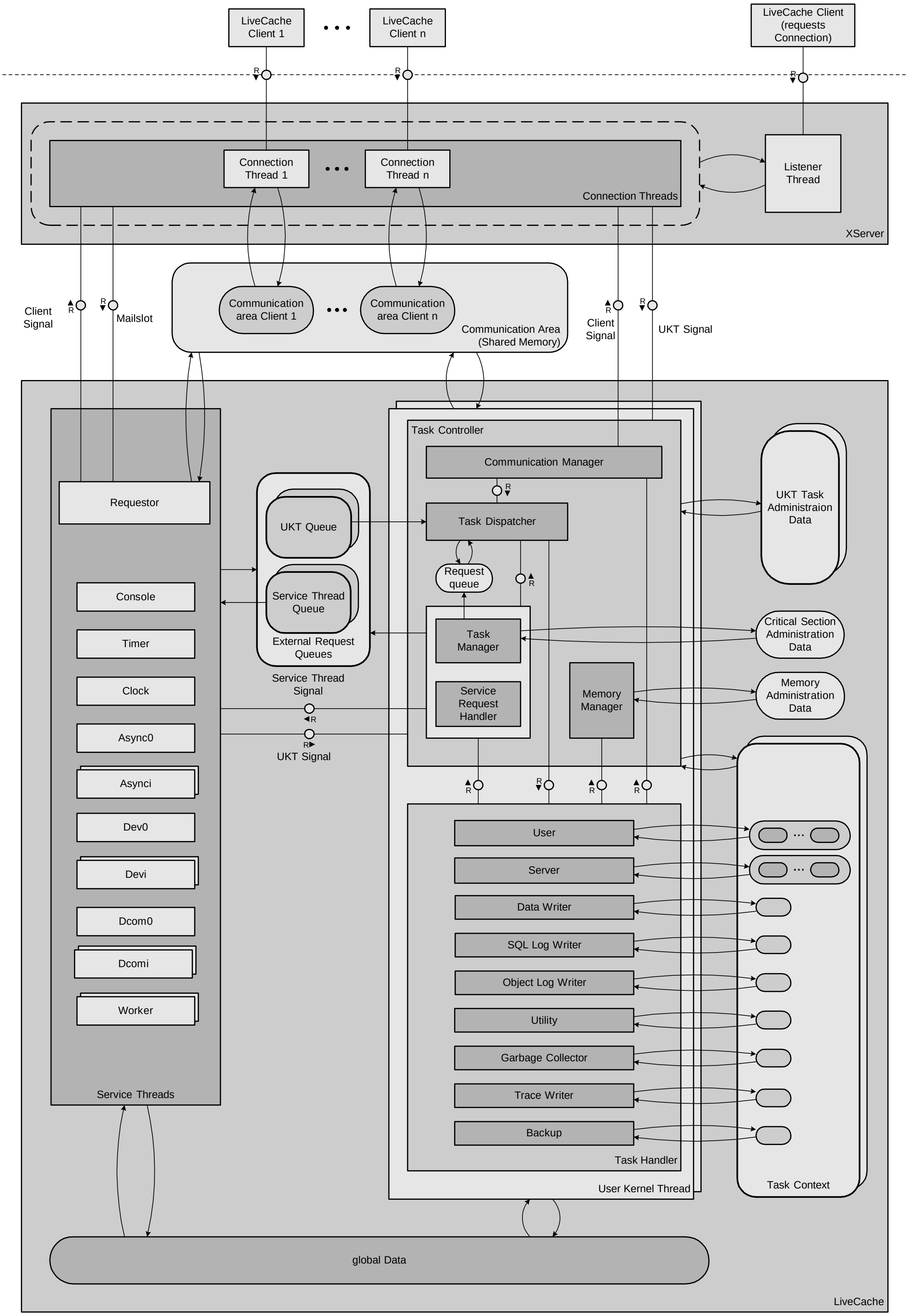
Block Diagram	1999/05/05
Figure A1 Client Communication Layer	
LiveCache	



Block Diagram 1999/05/04

Figure A2 LiveCache Basis Layer

LiveCache



Block Diagram	1999/05/05
Figure A3 Process and Thread Architecture	
LiveCache	