Generic Object-Oriented Database System

Konstantin Knizhnik

March 1, 1999

Contents

List of Tables

List of Figures

1 Introduction

GOODS is Generic Object Oriented Database system using active client language independent model of server-client interaction. All application logic is implemented and executed at clients while server is responsible only for storing and retrieving objects, handling transactions, object locks, garbage collection, database backups and recovery. Metaobject protocol is used in implementation of client application interface to database to provide transparent and flexible interaction with database. "Generic" in abbreviation GOODS stands for capability to extend system to handle almost all possible object access and synchronization strategies. Using "aspect oriented" approach makes it possible to changes object access management policies without affecting application code. Notation and ideas of "aspect-oriented" approach was taken from the works of Kickzales. Different strategies, such as standard serialized transaction model or optimistic model can be used as well as models fitting specific needs of concrete application.

GOODS is fully distributed database. Database consists of a number of storages, each of them can be located at different nodes of network. Storage is controlled by storage server, which is responsible for handling database requests relevant to this storage as well as for interaction with other servers to perform database wide operations. Client can work with arbitrary number of databases each time. Each persistent object is stored in concrete storage and it's location can't be changed. Location of object within database is mostly transparent for client (client should not worry about in which storage object is situated), but it is possible for client to attach object to concrete storage. Global database operations (operations affecting more than one storage in database) such as global transaction commit, garbage collection and object locking are handled automatically by database system using various algorithms to synchronize activities of each storage server. Online database backup and scheme modification are possible without stopping client working with database. Lazy object conversion approach is used for implementing scheme modification. Different client can simultaneously work with different versions of class definitions.

GOODS was designed to achieve maximum performance of handling object requests. Various algorithms of caching and prefetching objects and storage pages are used to reach this purpose. By splitting work into separate threads of control GOODS makes it possible to use all benefits of parallel execution especially at multiprocessor architectures. Specially designed library provides system independent interface for multitasking (classes and methods for creating and synchronization of tasks, including such synchronization primitives as mutexes, events, semaphores). This library is used to implement client part of database interface as well as GOODS database server itself. Using of metaobject protocol for handling access to object makes it possible to specify specific algorithms for object caching and synchronization of concurrent accesses, allowing to reach the highest level of performance for concrete application.

2 GOODS server

GOODS storage servers is divided into several components each of them is responsible for it's own task and interacts with other components by means of strictly defined interface. This components include storage memory manager, transaction manager, page pool manager, object access manager and class information manager. Such module design makes it possible to choose different implementations for each component (most suitable for concrete application) and also to investigate efficiency of various database management algorithms and strategies. Below is short description of each component.

2.1 Storage memory manager

This manager is responsible for allocating and deallocating storage memory and object descriptors. In GOODS indirect access model is used for interobject references. Each object has unique identifier (opid) and there is table of descriptors providing mapping of object identifiers to physical offset within storage file. This approach makes it possible efficient implementation of "become" operator which changes type of concrete object. In GOODS descriptor of object contains information about object class, size and location in the storage file. Object descriptors table is implemented using mapped on memory file. Allocation of storage memory in current implementation is done by using bit memory map. Each bit of this map corresponds to quantum of storage memory. Current size of the quantum is 32 bytes. Storage bit memory map is itself stored in mapped on memory file.

Garbage collection is done using variation of standard mark-and-sweep algorithm with some extensions for synchronization of GC processes at different storages. To perform GC one of storage servers is chosen as coordinator and is responsible for initiating GC processes at each server and determination of global end of mark phase at all servers.

GC can be initiated either when size of allocated objects exceed some predefined threshold value or after some period spent by system in idle state. To initiate GC each server sends request to coordinator. If coordinator decides that GC can be started, it poll other servers in database whether they are ready to start GC. When server receive such request from coordinator and sends acknowledgment to start GC, server change it's state to "PRE-GC". If acknowledgments are received from all servers coordinator broadcasts to all servers request to start mark stage of GC. Otherwise (if some server is not available or previous GC iteration still not finished at one of the servers) server broadcasts "CANCEL" request to all servers to make them exit from "PRE-GC" state.

Each storage server locally performs GC starting from storage root objects. Set of roots consists of storage root object and instances of objects loaded by client at the moment of starting GC. Objects which are already scanned considered to be marked with "black" color, while objects referenced from "black" object but not yet marked are considered to be "gray". All other object are "white".

When GC process finds out reference to object from other storage it sends message to this server with this reference. To make this exchange of references more effective, buffering of external references is done in GOODS. Before references will be send to client it is placed in export buffer (if such reference is not already present in buffer). When buffer is full, it is sent to destination server. Server maintains separate export buffer for each other server.

When there are no more "gray" objects in storage, server sends message to coordinator containing a vector of numbers of messages with external references sent and received by this server from other servers. Coordinator maintains matrix of all such vectors. To determinate global end of mark stage coordinator checks the following condition:

$M[i,j].import = M[j,i].export$, for all i, j

where $M[i,j].import$ is number of external references received by server i from server j, and $M[j,i].export$ - number of messages sent by server j to server i.

If this condition is true, then mark phase is completed at all servers and coordinator sends messages to all servers to initiate sweep phase of GC. While sweep phase all "white" objects are deallocated. The same approach is used to collect unused version of object classes.

To prevent deallocation of objects instances of which were just allocated and not yet committed by some transaction, all such instances are marked as "black" before sweep phase. GC process is running in background without affecting normal functioning of database. If object is changed during GC, difference of sets of references from old and new versions of object is calculated. (To make this operation more effective, not really set difference is calculated, but instead of it only references in near standing positions are compared. This makes it possible to handle most common cases: references are is not changed or are shifted due to insert or remove from array operation.) List of references from all old versions of object is kept until the end of GC. PRE-GC state is necessary to synchronize moment when all servers start to keep information about old versions of objects. So GC at each server will see all references between objects which were available at the moment of GC initiation. If some objects are changed by transaction, GC will have no access to this objects until transaction will be committed and all this objects are updated.

As far as different clients can have different version of objects, storage server should remember all versions of object used by active clients. This information can be removed when all clients update their instances of object and object is scanned by GC (if GC was active when object was modified).

Sweep stage is performed incrementally, making it possible to allocate new objects in storage while sweep process is active. System crash can cause some leak of memory due to garbage in memory allocation bitmap (for example space allocated for extended objects by transactions which were active at the moment of crash). Usually this leak of memory is very small (if any) and it is more significant to recover server as fast as possible. But it is possible to completely reconstruct bitmap after recovery (clean bitmap and then mark only space used by objects present in object index).

Allocation of object descriptors is performed using list of free object descriptors. As far as system fault can leave list in inconsistent state special checks are used in all operations with the list. If list is corrupted it will be completely rebuild at sweep stage of next GC (in normal situation GC only appends deallocated descriptors to this list).

Usually objects are allocated in file continuously (unless bitmap is fragmented), so object's body can cross page boundary. Usually continuous allocation provides good performance, because objects which are created together will be with big probability accessed together. So continuous allocation of objects will reduce page trashing. But for some classes of objects is preferable to place each object at separate page. B-tree page is an example of such kind of objects. As far as page size in GOODS is application dependent parameter which can be changed in any moment, the only thing we can ask memory manager to do with such objects is to align it's position in file on object size boundary (more precisely, on pow of 2 which is greater or equal than object size). By such alignment we can guaranty that object will never cross page boundary unless page size is smaller than object size.

2.2 Transaction manager

Transaction manager is responsible for handling all database updates as atomic and recoverable actions. Transaction can be either local (only one server participates in transaction) or global. In last case two-stage transaction commit protocol is used to guarantee global database state consistency. When global transaction is committed one of the servers participated in this transaction (one with lowermost identifier) is chosen to be coordinator. Coordinator assigns to transaction global identifier, writes it in global transaction history file and sends it back to the client. Client then sends local parts of transaction to all servers participated in this transaction together with identifier of coordinator and global identifier of transaction assigned to this transaction by coordinator. Coordinator then waits for response from all servers participated in this transaction (stage I). Receiving it's part of transaction server writes it to transaction log and sends message to coordinator reporting that he is ready for global transaction commit. When coordinator receives acknowledgements from all servers involving in transaction (stage II) it marks transaction in global transaction history file as globally committed, sends messages to all servers to finish transaction, flush all transaction changes in database and send response to client about successful transaction commit. Receiving such message from coordinator each other server participated in transaction will copy all object modified by this transaction in storage and release locks set by this transaction. So it is possible that client initiated transaction will receive reply from coordinator before this transaction is completed at all servers involved in transaction. But because commit of transaction is handled by client agent at server, server will not handle other requests from this client before transaction completion.

This protocol of transaction commit has uncertainty period, during which time the protocol might not be able to complete if coordinator fail. The participants must wait until the coordinator recovers before committing or aborting their local transactions. During this time, the local data accessed by transaction is locked to ensure serializability. When server is in uncertain state it periodically resends request to coordinator to obtain status of transaction until response will be received.

If at stage I coordinator receives "aborted" message from one of the servers or if timeout of waiting reply is expired then coordinator makes a decision that transaction is globally aborted, marks this transaction as been aborted in global transaction history log and sends abort messages to all servers participated in transaction and to client initiated this transaction.

While recovery from system fault server reads transaction log and checks whether transaction is local or global. If transaction is global then server asks coordinator of this transaction for status of this transaction.

Coordinator is looking in global transaction history log to check status of global transaction. If transaction is marked as globally committed, then coordinators sends "committed" reply to the server and this server performs recovery of it's local part of the transaction. Otherwise, if transaction is marked as "aborted" or if there is no entrance with such global transaction identifier in the global transaction log file then coordinator responds to the server with "aborted" message and server skips this transaction in it's log. If coordinator is not available at this moment then server has to wait until coordinator will be restarted.

When one of participants is failed after saving local part of transaction in the log but before sending "ready-to-commit" message to coordinator, it will ask coordinator for transaction status while recovery. If coordinator is still waiting for acknowledgement for transaction commit from servers participated in transaction (timeout is not still expired), it can notice that request for transaction status refers to the active transaction. In this case coordinator can treat "get-transaction-status" request as acknowledgement from this server to commit transaction (if server asks for transaction status it means that it has successfully save local part of transaction in the log and is ready now to recover it). So coordinator doesn't sends reply for this request immediately but instead of it continues with processing of transaction. Reply will be send to participants latter when transaction will be either committed or aborted.

When objects are restores from transaction log after crash, we should be careful with "zombie" objects - objects which were already deallocated by GC. Space of this objects can be reused for newly created objects. When we restore such "zombie" object from log it will override new objects allocated in the same memory space. That is not a problem - new object will be also present in log and will be restored later. But now we have several object descriptors refer to the same memory location. As far as there should be no references to old object (which was once collected by GC), it will be deallocated at next GC iteration and his space will be freed. But we also deallocate space of new objects which can be still accessible!

To avoid such dangerous behavior we keep track of address ranges occupied by all restored objects during recovery procedure. If ranges of some two object overlap, then older one is removed (spaced used by this object is deallocated and descriptor is inserted into the list of free descriptors). We use special kind of binary tree for effective finding of overlapped regions (log can contain a big number of objects and naive algorithm with complexity $N * N$ is not acceptable). Each level of this tree represents range which is power of 2. The depth of this tree is not much then 64 (offset in database file is 8 byte). So complexity of checking ranges of all restored objects is $C * N$, where C is some constant.

When storage is recovered after the crash, file memory allocation bitmap is cleared and all used objects (addressed by object index) are marked in this bit map. If some of the bitmap bits corresponds to the object space are already marked and object is not a one recovered from transaction log, then we have inconsistency in the object index. This inconsistency can be caused by garbage collector which free unreferenced object, reuse it's space for some newly allocated object, but index page with this descriptor was not flushed to the disk before the crash. Fortunately if one of two overlapped objects is really accessible from the storage root, then it should be in transaction log and so it was already recovered. So in this case we can just remove descriptor of the second objects. If both of the objects are not recovered from the transaction log, then both of them are inaccessible and we can safely remove one of them to reestablish consistency of object index.

When size of transaction log exceeds some limit value, checkpoint process is started by transaction manager. All modified pages and file buffers are flushed to the disks, afterwards log is truncated and logging started from the beginning of the file. New sequential number is written at the beginning of the log. This sequential number is incremented after each checkpoint and database close operation. This sequential number is included in each transaction record written to the log. Only that records which sequential number is equal to sequential number of the log will be restored while recovery after fault. When checkpoint is completed, server sends messages to all possible coordinators of global transactions in which this server can be involved (servers with identifiers less than identifier of this server) informing them about checkpoint completion. By receiving such message from server i coordinator can remove from global transaction log all entries relevant to the server with sequential number less than current sequential number of server i minus one. Subtracting one is necessary because transactions with this sequential numbers can be saved by backup and retrieved by restore procedure.

Transaction manager is also responsible for managing online backup procedure. Two types of backups are now supported by GOODS server: snapshot backup and permanent backup. Snapshot backup allows to produce snapshot of storage state. If disk failure will happened then it is possible to restore locally consistent state of storage for the time of backup completion (if global transaction are used, then global state of database can be not consistent because backup is local to storage server so snapshots at each server are not synchronized). Permanent backup process can be used to guarantee global database recovery in any moment of time. It is assumed that log files can not be corrupted (they can be placed on some reliable media - for example RAID disk massive). When backup process is active checkpoints are delayed till the end of backup. To avoid delays in database functioning due to waiting of backup completion, limitation for log size should be set to such value, which can guaranty that backup can be completed before this limitation is reached. Backup can be started by triggering one of two conditions: timer signal or size of log becomes greater than some specified value. Backup process first saves current contents of storage file, storage memory allocation bit map and object descriptor table. Then it copies to the backup file records from transaction log until it reaches the last one (records can be added to transaction log while backup is in progress so synchronization is required). When last record is reached backup considered to be completed and checkpoint procedure is invoked to truncate transaction log. After backup completion operator has to switch backup media (tape) and make schedule for next backup - specify timeout and(or) log size when next backup should be started.

The difference between permanent backup and snapshot backup is that snapshot backup has to save only that part of transaction log file which was written at the moment when backup completes copying of storage file and memory manager tables. Backup should save only information about transactions completed before this moment to provide consistent snapshot of the storage. Checkpoint procedure is not forced by snapshot backup.

Operations of writing transaction into the log should be performed synchronously, to guarantee that after completion of this operation it will be possible to recover transaction in case of system fault. Total throughput of database system can be significantly increased if separate synchronous writes are combined into single write operation. Also performance can be increased if size and position of written data are aligned to the operating system disk block size. In this case operating system should not previously read contents of modified block from the disk. This algorithm is encapsulated into "file" class. This abstract class provides operating system independent access to files, guaranteeing correctness of concurrent access operations to the file. Implementations of this class for different operating system use system specific advanced calls (such as writev for Unix or overlapped IO for Windows-NT) to achieve maximal performance. To provide merging of concurrent synchronous write requests, file implementation uses two buffers in cyclic way. While one buffer is writing to the disk all other write requests are collected in other buffer. When write operation for first buffer is completed, roles of buffers are changed and following write requests will be placed in first buffer. Size of buffer is aligned to the size of operation system disk block, so no overhead of reading content of modified block is present. In case of using gathered IO functions (such as writev) it is possible to avoid coping of all written data to buffer, but instead of this construct vector of segments to be written on disk.

It is possible in GOODS to switch off synchronous writes to transaction logs (use normal buffered write requests) if operating system can guarantee that all modified buffers will be flushed to disk before system halt. That is possible if UPS is used to prevent power faults and operating system itself is considered to be enough reliable. Performance can be dramatically increased in this case.

2.3 Page pool manager

Page pool manager provides efficient access to storage file. To improve IO operation performance, page pool is used for caching most recently used pages and buffering IO requests. Special synchronization policy allows different clients to perform read/write operations in parallel. Algorithm is similar with one used by operation systems to control access to file buffers. Each page has BUSY and WAIT flags and event object on which tasks accessing this page can wait. When page is read from file BUSY flag is set, so any other task trying to access this page will set WAIT flag and sleep on correspondent event object. When read operation is completed, WAIT flag is checked and if it is set then event object is signaled awaking all waiting tasks. While page is used by some tasks (they copy data to or from it) USED counter in page header is not zero. This counter prevents page replacing algorithm from throwing this page from the cache.

The storage page size is application dependent parameter and can be changed without affecting storage data file. By default the operating system page size is used as a value for this parameter.

2.4 Object access manager

Object access manager controls locking of objects, maintains information about instances of objects loaded by clients, send notifications to clients when object is modified and also synchronize access to objects by different components of storage server.

Two kinds of locks are supported: exclusive and shared. Only one process can lock object in exclusive mode and several - in shared. Lock requests can be either blocked or not-blocked. In first case process requesting lock will be blocked until requested lock is granted. In case of non-blocking requests if granting lock is impossible then negative answer is returned immediately. In GOODS "honest policy" of granting locks is used: lock requested first will be granted first. So even if requested lock is compatible with current locks set on the object, this request can be blocked because there are more blocked lock requests for this object. There is only one exception from this rule: shared locks can be upgraded to exclusive despite to presence of other blocked requests.

As far as different components of storage server can access the same object simultaneously, some synchronization mechanism is needed. Object access manager provides methods for two type of object accesses: reading and writing, implementing "multiple readers single writer" discipline.

Object access manager maintains for each object list of object instances loaded by client processes. When object is changed this list is scanned and all processes having instance of this object (except one which has modified this object) are send a notification message. For correct garbage collection it is necessary to know about all objects and references from this objects presented in client processes. As far as object access manager has this information, garbage collector process requests information about all such references from object access manager during mark phase and before sweep phase.

As far as client can have deteriorated version of object and follow references from this object instance, it is necessary to prevent deallocation by garbage collector of the object, referenced only from deteriorated instances (so no more references to this object are present in database). When object is modified, object access manager compares set of references in old and new version of the objects. All references in old version of the object which are not present in new version are inserted in "old-reference-list", explored by garbage collector. This references are kept in this list until all clients will update their instances of objects.

2.5 Class information manager

GOODS uses "active-client" model of client-server interaction. All application logic is programmed at client application and servers are responsible only for fetching and storing objects and synchronizing access to them. So server is not need to know to much about contents of objects, it views an object as the array of references and raw data (knowledge about object references is necessary to perform garbage collection). So object is stored at server in system independent format (big-endian byte order, 6-bytes references containing object storage identifier and number of the object within storage). To simplify for server work with objects (and as a result - increase server performance) objects are stored at server in format with all references placed before any other data.

GOODS server stores information about classes of all objects allocated in this storage. Class information includes class name, size of fixed and varying part of class, number of references in fixed and varying parts and also information about each object field: names, type and offset within object. This information is used by server only to calculate number of references in the object while garbage collection and building closure of objects for sending to client. But storing complete information about the class definition at server is significant to provide work of clients with different database schemes (class definitions). GOODS allows online modification of database scheme without termination of clients working with database. Moreover one client can use old version of class definition, while another a new one. Such lazy object conversion strategy is really necessary for system with a lot of different client's applications and working time $7x24$.

When client is loading object from server it looks at its class identifier. If client had already loaded object of such class then mapping between database class and local application class was already set. Otherwise client sends to the server request to provide information about class with such identifier. The server responds with full class description. The client then looks for local application class with the same name. If no such class is found application is abnormally terminated. Otherwise client compares signatures of storage and application classes. If signatures are the same then client just establish new mapping between storage class identifier and application class.

If signatures are not equal then it is considered that client has new version of the class and special class descriptor for converting object from old representation to the new one is constructed. Conversion procedure looks for definition of all components in old class description and tries to find fields with the same name in new class description. If field with the same name is present in boss class definition, then conversion procedure checks whether conversion between old type of the field and new one is possible. Now the only restriction for types compatibility is that reference type can be only assigned to references type. All other kinds of conversion (real-¿integer, integer-¿real, fixed array of scalars-¿scalar, varying array-¿fixed array...) are possible. Certainly some of this conversion can cause loss of data. Then client sends new signature of class to the server and receives new storage class identifier assigned to this class descriptor by server. When object of such class is loaded from the server it is converted to new class format and new storage class identifier is assigned to this object. If such object will be modified and send back to the storage it will be saved in new class format. So lazy approach is used to perform object conversion when class definition is changed.

When client is going to store object in storage (as a result of making reference to this object from another persistent object) it checks whether this object has assigned storage class identifier. If none then client sends to the server description of object class and waiting for response containing storage class identifier assigned to this class. When server receives such request from client, it looks through definitions of all classes in the storage and if class with the same signature is found replies with it's identifier. Otherwise new class definition is added to storage and new assigned storage class identifier is returned to the client.

As far as there are can be a lot of object class modifications, garbage collection procedure is also necessary to collect unused classes. When mark stage of garbage collector is performed classes of all "black" objects are also marked. At sweep stage class information manager is asked to remove all unmarked classes. As far as class can be registered by application but object instances of this class are not yet created at this moment, deletion of this class definition is delayed to the moment when all client processes working with this class descriptor will be terminated. To implement this strategy of delayed class deallocation all clients are assigned successive numbers at the time of connection to the server. Each class descriptor stores the maximal client identifier accessed to this class (when client sends request to provide class identifier for concrete storage class descriptor, identifier of this client is compared with client identifier stored in this class descriptor and if first is bigger then it replaces client identifier stored in this class descriptor). Class descriptor can be removed when there are no object instances of this class in storage and there are no more clients with identifiers less or equal to class identifier stored in this class descriptor.

Class descriptors are stored in storage as other objects but special range of object identifiers is used for class descriptor objects. In GOODS class storage identifier is stored in one word (two bytes) of object descriptor, so the maximal number of classes in the storage is limited to $2^16 = 65535$. So first 2^16 entries in object descriptor table are reserved for class descriptors. Creating new class descriptors and updating of existing ones is performed using standard transaction mechanism. To improve performance of class descriptor lookup, class information manager maintains array of copies of all class descriptors in memory and also has hash table for fast search of class with specified name.

2.6 Optimization of object loading

When client application accesses persistent object which is not present in client cache, database interface sends request for loading object to the server where this object is located. As far as transmission data through network is relatively slow operation and moreover only small portion of total time is used to transfer data itself (it is not so much difference in time between sending 1 byte or 100 bytes of data) there are two ways of increasing application performance. First is to increase size of object cache and improve cache management algorithms. We will speak about this approach latter. And another opportunity to increase performance is to predict which objects will be retrieved by client and to send in reply message not only requested object, but all objects which will be with big probability requested by client in near future. So when client accesses such object, client database interface finds that it is already present in object cache and no request to server need to be send. Certainly it is very difficult task to predict which objects will be retrieved by application in near future. Looks like no single best solution exists for this problem.

There are several main approaches to determinate set of objects which should be send to client. One alternative is to ask programmer to explicitly specify clusters of objects. If one objects from the cluster is requested, then all objects from this cluster are transmitted to the client. Disadvantage of this approach is that it violates transparency of manipulation with persistence objects. Another approach is to maintain statistic of requesting objects by clients. So if server knows that after requesting object A client with high probability will request object B, then object B can be send to client together with object A. Disadvantage of this approach is that it consuming a lot of server memory and CPU. Also while this approach provides very best result for some applications it can be inefficient for other applications. And the third approach is to try to construct some kind of object closure including in this closure all (or some of) objects referenced from requested objects. The size of closure should be limited to avoid increase of network traffic due to transfer of useless data.

In GOODS server third approach is currently used (certainly other approaches can be easily implemented and tested by redefinition of one method). When object is requested by client, server includes in reply all objects which satisfy to three of following conditions:

1. object is directly accessible from requested object,
2. client doesn't have this object,
3. total size of all objects in reply is limited to some predefined constant.

3 Application interface to database

Application interface to database is divided into two parts: application dependent and application independent. Application independent part is represented by abstract class `dbs_storage` which contains reference to abstract class `dbs_application` which declares methods handling database requests to application (notification about object modification, disconnect handler). Class `dbs_storage` declares methods for fetching, locking and unlocking objects, retrieving and storing class descriptors, manipulations with transactions,... Implementations of this class are responsible for sending requests to server and receiving replies and notifications from the server. This class is not responsible for synchronizing access to the storage and for allocation, unpacking and deallocation of object instances in clients application. Instead of this implementation of `dbs_storage` just place object instance or class descriptor loaded from server in specially supplied buffer and future operations with this buffer are performed by application dependent part of database interface. Application dependent part of database interface implements `dbs_application` protocol and contains methods accessing `dbs_storage` and providing synchronization of storage access as well as object packing and unpacking. Also handling of storage notifications is performed by methods defined in this part of database interface.

Interface with database is organized in such way that it is possible for different applications written using different languages to work with the same database. In current version of GOODS only interface for C++ is provided. This is because C++ is the most popular object oriented language now, and also providing most performance efficient executables. Interface to Java language is planned in near future.

Different classes of application may have very different requirements to the database system. For example for banking applications standard serializable transaction model may be most relevant one. Applications in such system should have an illusion of monopoly work with database: transaction should not see changes made by other transactions of other clients. It is isolation model of database organization. But for office document flow controlling system cooperation model may be more preferable. In that model applications cooperate with each other and changes made by one application should be visible by other applications.

As far as it is not possible to incorporate all possible strategies in single database system, then generic (or universal) object oriented database should have facilities to extend it's functionality and behavior depending on applications requirements.

Modern programming systems have to deal with a lot of different aspects: user interface, memory management, multitasking, database access, security... A traditional approach of dividing application into modules will not work if each module will be responsible for all of this aspects. If for example logic of synchronization access to objects will be scattered through the all application, then it will very difficult to understand such code, debug or modify it. To make application more clear, simple and extensible it will be goods idea to separate code responsible for different aspects of system behavior. This separated aspects can be combined together at compilation phase (or at runtime) using metaobject protocol approach. According to GOODS interface with client applications the following aspects of the system are implemented by metaobjects:

- Intertask synchronization of object access
- Synchronization of object access by different database clients
- Handling of database transactions
- Managing of client's object cache

Lets look at this aspects more closer:

3.0.1 Intertask synchronization of object access

Modern applications very often has to deal with a lot of concurrent jobs: handling user interface, retrieving data from database, performing some background recalculations... It is very difficult to implement such system without using multitasking. Multitasking can be either explicit or implicit. Implicit multitasking is more transparent for programmer (for example each method invocation can be considered as separate thread of control), but it is not so flexible as explicit multitasking and as far as aspects of it's implementation are hidden from programmer, some kind of unexpected by programmer behavior can take place. That is why in GOODS we decide to use explicit multitasking model where programmer has to create and synchronize tasks explicitly using such synchronization object as mutexes, events, semaphores. Synchronization of access to the object by different tasks forms separate aspect of system behavior which is implemented by means of metaobject protocol.

The default policy used in GOODS for intertask object access synchronization is mutual exclusion model. Only one task can access object at the same time. Each GOODS object has associated monitor object which synchronizes access to the object instance by different tasks. When method of the object is invoked by some task, monitor associated with this object is locked to prevent other tasks from using this object until this task returns from invoked method. Nested calls to objects are also allowed. This is simple and powerful strategy but it has some serious disadvantages, one of them is possibility of deadlock - mutual blocking of several tasks.

This model of intertasking object access synchronization is similar with one used in Java (with all methods explicitly considered to be "synchronized"). Each GOODS object also has two methods "wait" and "notify". When a task, executing in a monitor needs to wait for another task to do something, it calls wait(). This causes task to unlock the monitor and go to sleep. Since monitor is unlocked, another thread can enter monitor and supply the information the first task is waiting for. The task signals the waiting task by calling notify(). Once the first task is signaled it wakes up and begins waiting for the monitor to become available. When the second task finishes its processing, it unlocks the monitor, allowing the first task to reacquire the monitor and finish what it started.

Unfortunately, there is one serious problem with this signaling mechanism, which can cause deadlocks. If method of object O1 invokes method of object O2, which in it's turn calls wait() method, then only the monitor of object O2 is unlocked, while the monitor of object O1 remains locked. Something that looks tempting at first is to modify behavior of metaobject so that calling wait() unlocks all monitors a task has acquired. This would be a mistake. It would imply that anytime you call a function that could in turn potentially call wait, you cannot guarantee that another task won't change the state of the object.

As far as monitor objects takes some space it is very space consuming solution to have separate monitor object for each application object. Instead of this "turnstile" of monitor objects is used in GOODS. When object should be locked and it has no attached monitor object, then new monitor object is taken from the turnstile (if there are no more free monitor objects, then turnstile is extended). When object is unlocked, monitor object continues to be attached to this object, but it can be taken away and used for another object. So the total number of monitor objects doesn't exceed maximal number of simultaneously accessed objects and there is high probability that when object is accessed it already has attached monitor object.

3.0.2 Synchronization of object access by different database clients

There are two main approaches to object access synchronization in database: pessimistic and optimistic (certainly some combination of this two approaches can be used). With pessimistic approach object locks are usually used to prevent other clients from accessing the object. This approach guarantees that object changes can not be lost. Main disadvantages of this approach are possibility of deadlocks and reducing concurrency. Optimistic approach can be successively used if possibility of conflict (simultaneous modification of object by different clients) is small and transaction can be easily restarted. With this approach check for correctness of object access is made only when transaction is committed. If some of the objects touched by transaction has been changed by some other transaction then conflict takes place and current transaction should be restarted. There are a number of different metaobjects in GOODS database interface implementing different variations of this approaches. Below is a picture illustrated hierarchy of this metaobjects. Description of this metaobjects can be found in "mop.h" header file.

```
Abstract Metaobject
   Basic Metaobject
      Optimistic Metaobject
                    Repeatable Read Optimistic Metaobject
      Pessimistic Metaobject
                    Lazy Pessimistic Metaobject
                    Repeatable Read Pessimistic Metaobject
```

It is possible to achieve higher level of concurrency between different clients if semantic of concrete object class is explored. For example for "queue" object PUT and GET methods can be executed concurrently without mutual exclusion (in spite of they are both mutators) as far as queue is not empty. It is possible because this methods work with different ends of queue. Using of specific metaobject protocol for such objects can get advantage from this knowledge of object semantic.

3.0.3 Handling of database transactions

All changes in database should be made by means of consistent and atomic sequences of operations - transactions. Transaction transfers database from one consistent state to another. There are a lot of different transaction models: flat transactions, nested transactions...() Simply speaking all this models have different answers for two questions: when transaction should be committed or aborted and when changes made by transaction should become visible for other transactions. As far as different application may require different transaction models we want to left answer on this questions for method of metaclass. So by definition of own metaobject programmer can introduce it's own model of transactions.

Default implementation of transactions in GOODS implicitly opens nested transaction each time GOODS object is accessed and commits all nested transactions when control returns from last invoked method. So programmer should not worry about definition of points where to open and to close transaction (but it is possible to explicitly open and close nested transaction and so to create long-live transaction). Transaction can be aborted, in this case all modification of persistent object are discarded. Such implicit transaction scheme is highly compatible with idea of transparent database interface and also is more error safe.

3.0.4 Managing of client's object cache

As far as client application has to send requests to server for every accessed object, performance of application mostly depends on efficiency of object caching strategy. Usually standard LRU algorithm is used for replacing entries in cache, but for database this discipline is not always good choice. Application accessing database frequently performs search among big number of objects. With traditional LRU cache replacement algorithm this scanned objects (most of them will not be more accessed in near future) will completely replace all cache entries by throwing away all other objects.

To avoid such undesirable behavior special modification of LRU algorithm is used in GOODS. Object cache is divided into two parts: frequently used objects (FUO) and objects used only once (OUOO). Both of parts are controlled by ordinary LRU discipline using double linked lists (object is excluded from the list when it is accessed and inserted at head of the last after the end of access). But if object is taken from OUOO list and it is not at the head of the list then object is considered to be frequently used one and reattached to the FUO list. (As far as object is present in OUOO list and is not at the head of the list then we can make a conclusion that it was already accessed by some other place in the program and looks like it can be accessed once again). So object scanned while database search can not replace objects from FUO part of the cache. But GOODS leaves opportunity for programmer to define own cache managing policy because cache is also controlled by metaclass methods.

Current GOODS database interface consists of some kernel database requests (such as set lock, open transaction, load object...) and a number of basic metaobjects, implementing most common models of object access organizations. Deriving from this basic metaobjects it is possible to create specific metaobjects to handle specific requirements of concrete application. Usually most of the work can be done by methods of base metaclass and only few things should be redefined or added.

3.1 System abstraction layer

3.2 GOODS interface for C++ language

GOODS supports the following scalar types as components of database classes:

```
char           1 byte character
nat1  unsigned 1 byte integer
int1    signed 1 byte integer
nat2  unsigned 2 bytes integer
int2    signed 2 bytes integer
nat4  unsigned 4 bytes integer
int4    signed 4 bytes integer
nat8 unsigned 8 bytes integer
int8    signed 8 bytes integer
real4   4 bytes ANSII floating type
real8   8 bytes ANSII floating type
```

It is better to use these type aliases instead of native C types to provide portability of your application (for example type long can be 4 bytes at one system and 8 bytes at another). References to another GOODS objects are supported by means of "smart pointers" implemented by template class **ref**:

```
class A;
ref<A> ra;
```

It is possible to construct arrays and structures of specified above atomic types:

```
   char str[10];
   struct coord {
int x;
int y;
   };
   coord points[10];
```

It is possible to specify classes with varying size of object. Such class should have one varying component size of which is determined at the time of object creation:

```
  class person : public object {
  public:
       char name[1];

       static tree_node* create(char* name) {
           int name_len = strlen(name);
           return new (name_len) person(name, name_len);
       }
       METACLASS_DECLARATIONS(person, object);

   protected:
       person(const char* name, size_t name_len)
       : object(self_class, name_len)
       {
           memcpy(this->name, name, name_len+1);
       }
    };

    field_descriptor& person::describe_components()
    {
        return VARYING(name);
    }
```

Class can have only one varying component and it should be the last one. Classes with varying components are used in GOODS C++ interface to implement arrays and provide efficient way for representing classes with constant (immutable) string identifier (like person and name).

As far as in most of implementation of C++ metaclass information is not available and database needs to know format of objects, programmer has to specify this information manually. To make this work more easy a number of macros and functions are supported in GOODS interface for C++.

GOODS supports persistency only for objects of persistent capable classes. Class is persistent capable if it is derived from GOODS "object" class and implements some methods and constructors needed by GOODS client library. Such class should have:

1. Specific constructor for initialising object when it is loaded from database;
2. Static component `self_class` containing class descriptor of this class;
3. Overloaded function "classof" returning class descriptor determined by static type of function argument;
4. Virtual method `describe_components`, which returns information about all components in the class.

All this components are declared by macro
`METACLASS_DECLARATIONS(CLASS, BASE_CLASS)`

A programmer needs only to implement the member function `describe_components`, which provides information about all class instance variables. To make process of class variables description more easy and error safe five special macros are provided:

`NO_FIELDS` method should return this value if there are no variables in class
`FIELD(x)` describing atomic or structural field
`ARRAY(x)` describing fixed array type
`MATRIX(x)` describing two dimensional fixed array type
`VARYING(x)` describing varying array

If there are structural components in class you should define function **`describe_field`** to describe components of this structure:

```
class B_page : public object {
    friend class B_tree;
    enum { n = 1024 }; // Minimal number of used items at not root B tree page
    int4 m;            // Index of first used item

    struct item {
        ref<object> p; // for leaf page - pointer to 'set_member' object
               // for all other pages - pointer to child page
        skey_t     key;

        field_descriptor& describe_components() {
    return FIELD(p), FIELD(key);
}
        inline friend field_descriptor& describe_field(item& s) {
            return s.describe_components();
}
    } e[n*2];

    ...
  public:
    METACLASS_DECLARATIONS(B_page, object);
};

field_descriptor& B_page::describe_components()
{
    return FIELD(m), ARRAY(e);
}

REGISTER(B_page, object, optimistic_scheme);
```

Macro `REGISTER` can be used to define default implementations for methods **`classof`**, **`constructor`** and to create class descriptor for this class:

```
REGISTER(CLASS, // name of the class
 BASE,  // name of base class
 MOP    // metaobject for this class
        );
```

It is possible for template class to be persistent capable, but programmer has to explicitly register different template instantiations in database. The usual way is to use typedef operator to create alias to concrete template instantiation and then register class with this alias name in the database by means of REGISTER macro.

Each storage has predefined root object. To change type of this abstract root object you should use "become" method. To find out if storage is already initialized special virtual method **`is_abstract_root`** is defined in class "object". This method returns true if type of the root object has not been changed yet and false otherwise.

Multitasking library requires some initialization before it can be used. So the first statement in the program (in main function) should be invocation of static method task::initialize. The single parameter of this method specifies stack size reserved for main thread of the program.

Usually sequence of steps to initialize storage looks something like this:

```
class my_root : public object {
  public:
    ...
    METACLASS_DECLARATIONS(my_root, object);

    my_root() : object(self_class)
    {
       ...
    }
    void initialize() const {
        if (is_abstract_root()) {
            ref<my_root> root = this;
            modify(root)->become(new my_root);
}
    }
};

int main(int argc, char* argv[])
{
    task::initialize(task::huge_stack);
    database db;

    char* cfg_name = new char[strlen(argv[1])+5];
    sprintf(cfg_name, "%s.cfg", argv[1]);

    if (db.open(cfg_name)) {
        ref<my_root> root;
        db.get_root(root);
        root->initialize();

... // do something with database

        db.close();
    }
}
```

You can find template of simple database application in file "template.cxx". All accesses to persistent objects should be encapsulated inside object methods. It is an error to pass reference to some component of object:

```
class text {
  public:
    char str[1];
    ...
};

main () {
    ref<text> t;
    ...
    printf("text; %s\n", t->str); // !!! ERROR
}
```

The reason of such restriction is that persistent object can be at any moment thrown away from memory by cache replacement algorithm, when there is no active method for this object. The right implementation for the sequence above is:

```
class text {
  protected:
    char str[1];
  public:
    void print();
    ...
};

void text::print()
{
    printf("text; %s\n", str);
}

main () {
    ref<text> t;
    ...
    t->print();
}
```

You should try to avoid direct access to object components whenever it is possible: it is better to make all object instance variables protected and encapsulate all access to them in object methods. Encapsulation makes you application simpler and more flexible, and with GOODS C++ interface it also increase performance, because access to self components within object methods requires not extra runtime overhead.

GOODS C++ interface provides library of some widely used container classes for efficient access to persistent data. The following subsections briefly describes these classes.

3.2.1 Dynamic arrays

GOODS interface for C++ provides template class for dynamic arrays. Parameter of the template should be either builtin primitive type (nat1, int4, real8...) or persistent object reference. It is important to notice, that template instances should be explicitly registered in database by REGISTER macro. The following array template instantiations are defined and registered in dbscls.h: ArrayOfByte, ArrayOfInt, ArrayOfDouble and ArrayOfObject. If you want to use array of some other component type you should first create type alias by means of typedef operator and then register this type in the database:

```
     typedef array_template ArrayOfShort;
     REGISTER(ArrayOfShort, object, pessimistic_repeatable_read_scheme);
```

Dynamic array template provides methods for direct access to array components:

```
T operator[](nat4 index) const;
T getat(nat4 index) const;
void putat(nat4 index, T elem);
```

Methods for getting number of components in the array, for copying and appending array components are also available. Dynamic arrays also implement stack protocol by providing such methods as Push(T value), Pop(), Top(). Methods insert(int index, int count, T value) and remove(int index, int count) can be used to add or remove elements from the dynamic array.

Class String is implemented as subclass of ArrayOfChar class and defines extra methods for strings manipulation.

3.2.2 Sets

GOODS class library provides CODASYL-like sets to represent on-to-many and many-to-many relationships between objects. The set consists of one owner and many member components. The set owner is represented by `set_owner` class and provides methods for inserting, removing member to/from the set and iterating through the set members. The set members are accessed through `set_member` class, which contains member key and virtual function to calculate short key representation (used in B-tree). Both `set_member` and `set_owner` classes contain pointer to attached object, so the object can be a member of some number of sets and an owner of some other sets at the same time.

3.2.3 B*-tree

B-tree is classical data structure for DBMS. It minimize number of disk read operations needed to locate object by key and preserve order of elements (range requests are possible). Also maintenance of B-tree can be done efficiently (insert/remove operations have log(N) complexity).

In classical implementation of B-tree, each B-tree page contains set of pairs ¡key, page-pointer¿. The nodes at the page are ordered by key, so binary search can be used to locate item with greater or equal key. In B*-tree, pointers to members are stored only in leaf pages of B-tree. All other pages contain pointers to child pages.

B*-Tree in GOODS is implemented as subclass of `set_owner` class. Pointers of leaf pages of B*-tree refer to objects of `set_member` class, which contain references to the objects included in `B_Tree`. Nodes of the B*-tree pages contain short form of key (currently `nat8` type is used), which can be calculated from the object key by virtual method of `set_member` class (usually it is just first 8 bytes of original key). Such structure allows objects to be included in several B-trees and also makes search operation more effective, because only small `set_member` objects are accessed during search (if there are several objects with the same value of short key). `B_Tree` class defines methods for inserting new objects in the tree, removing objects from the tree and searching objects by the key.

3.2.4 Hash Table

Class `hash_table` provides fast almost constant time access to the object by the key. GOODS class library for C++ provides implementation of non-extendable hash table operating with string keys. Hash table can be effectively used if upper limit for number of objects in hash table is known and doesn't exceed size of table more than several times and hash table can fit in operating memory.

3.2.5 H-Tree

H-Tree is combination of hash table and index tree. It can be used when size of hash table is too large to make possible represention of hash table as single object (as array of pointers). H-Tree first calculates normal hash key and then divide it into several groups of bits. First group of bits is used as index in the root page of H-Tree, second group of bits as index in the page referred from the root page, and so on... So if the size of the hash table is 1000003, than H-tree with pages, containing 128 pointers, requires access to three pages to locate any object. So total size of loaded objects is 128*6*3 = 2304 bytes instead of 6Mb if `hash_table` class with such size is used (size of reference in GOODS is 6 bytes).

3.2.6 Blob

Most of modern database applications have to deal with large objects, used to store multimedia and text data. GOODS class library has special class Blob to provide efficient mechanism for storing/extracting large objects. As far as loading large object can consume significant time and memory, Blob object allows scattering of large objects into parts (segments), which can be accessed sequentially. Moreover, Blob object takes advantage of multitasking model of GOODS and makes it possible to load the next parts of the Blob object in parallel with handling (playing, visualization,...) of the current part of Blob. Such approach minimizes delays caused by loading object from the storage.

3.2.7 R-Tree

`R-tree` provides fast access to spatial data. Idea of `R-Tree` is the same as in B-Tree — use hierarchical structure with high branching factor to reduce number of disk accesses. The R-tree is the extension of the B-tree for multidimensional object. A geometric object is represented by its minimum bounding rectangle (MBR). Non-leaf nodes contain entries of the form (R, ptr) where ptr is a pointer to a child node in the R-tree; R is the MBR that covers all rectangles in the child node. Leaf nodes contain entries of the form (obj-id, R) where obj-id is a pointer to the object, and R is MBR of the object. The main innovation in the R-tree is that father node are allowed to overlap. This way, the R-tree can guarantee at least 50Guttman. GOODS `R_tree` class is based on Guttman's implementation with quadratic split algorithm. The quadratic split algorithm is the one that achieves the best trade-off between splitting time and search performance.

3.3 Running GOODS applications

3.3.1 Database configuration file

To specify configuration of database you should create configuration file with the following format:

```
<number of storages = N>
<storage identifier 0>: <host name>:<port0>
...
<storage identifier N-1>: <host name>:<portN-1>
```

Storage identifier should be successive integer numbers used as indices in the array of storages. You can see examples of this configuration files: "unidb.cfg", "guess.cfg". In distributed environment configuration file can be accessed from the server computer using some network file system protocol (for example NFS) or can be replicated to client computers.

3.3.2 Server monitor GOODSRV

To run database you should first start all storage servers at each node of the net specified in configuration file. You can write server program yourself, but GOODS provides standard server implementation "goodsrv" supporting some basic monitoring functions. To run this program you should specify the name of database. which should be equal to the name of configuration file without extension (extension assumed to be ".cfg"). First line of this configuration file specifies number of storages in the database. All successive lines specifies locations of database servers. Each line consists of three fields separated by colon: storage identifier, host name and port number.

Parameters of GOODSRV can be specified in one of two files: "goodsrv.cfg" and "database.srv". First one specifies parameters common for all servers and second - parameters of the server of specific database. If some parameter is defined in both of the configuration file, then value of the parameter from "database.srv" is used. If some parameter is not specified in configuration files, then default values will be used. The following table describes all available parameters:

Parameter	**Type**	**Unit**	**Default value**	**Meaning**	**Set**
`memmgr.-init_map_file_size`	int.	Kb	8192	Initial size of memory map file. Increasing this parameter will reduce number of memory map reallocations.	-
`memmgr.-init_index_file_size`	int.	Kb	4096	Initial size of index file. Increasing this parameter will reduce number of index reallocations.	-

Parameter	**Type**	**Unit**	**Default value**	**Meaning**	**Set**
`memmgr.gc_init_timeout`	int.	sec	60	Timeout for initiation of GC process. Coordinator of GC will wait replies from other server for GC initiation request during specified period of time	+
`memmgr.-gc_response_timeout`	int.	sec	86400	Timeout for waiting acknowledgment from coordinator to finish mark stage and perform sweep stage of GC. If no response will be received from GC coordinator within this period, GC will be aborted at the server.	+
`memmgr.-gc_init_allocated`	int.	Kb	1024	Size of allocated memory since last GC, after which next garbage collection process will be initiated	+
`memmgr.-gc_init_idle_period`	int.	sec	0	If non-zero then specifies idle period interval, after which GC will be initiated. If memory management server receives no request during specified period of time, then GC process will be initiated.	+
`memmgr.-gc_init_min_allocated`	int.	Kb	0	Minimal size of allocated memory to start GC in idle state (see previous parameter). GC will be initiated only if idle period timeout is expired and more than `gc_init_min_allocated` memory was allocated since last GC	+
`memmgr.-gc_grey_set_threshold`	int.	refs	1024	Specify maximal extension of GC grey references set. When grey references set is extended by more than specified number of references, then optimization of order of taking references from grey set (improving references locality) is disabled and breadth first order of object reference graph traversal is used.	+
`memmgr.-max_data_file_size`	int.	Kb	0	If non-zero, then set limitations for the size of storage data file. After reaching this value, GC is forced and all allocation requests are blocked until enough free space is collected.	+
`memmgr.max_objects`	int.	objects	0	If non-zero, then set limitation for number of objects in the storage After reaching this value, GC is forced and all allocation requests are blocked until some object will be collected by GC	+

Parameter	**Type**	**Unit**	**Default value**	**Meaning**	**Set**
`memmgr.map_file_name`	string	-	`*.map`	Name of file with memory allocation bitmap.	-
`memmgr.index_file_name`	string	-	`*.idx`	Name of file with object index.	-
`transmgr.-permanent_backup`	bool.	0/1	0	Specifies whether permanent or snapshot backup type should be used. If snapshot backup type is used, then backup is terminated after saving consistent state of database and checkpoints will be enabled. Otherwise, if permanent backup type is used, then backup terminates and forces checkpoint after saving all records from transaction log. Permanent backup can be used to ensure, that storage can be restored after fault and loosing storage data file.	+
`transmgr.max_log_size`	int.	Kb	8192	Size of transaction log after reaching which checkpoint is started. After checkpoint completion, writing to the log file continues from the beginning.	+
`transmgr.-preallocated_log_size`	int.	Kb	0	This option forces transaction manager to preallocate log file and doesn't truncate it after checkpoint. In this case file size should not be updated after each write operations and transaction performance is increased about 2 times.	+
`transmgr.wait_timeout`	int.	sec	600	Timeout for committing global transaction. Coordinator will wait for replies of other servers participated in global transaction until expiration of this timeout.	+
`retry_timeout`	int.	sec	5	Timeout for requesting status of global transaction from coordinator. When server performs recovery after crash, it need to know status of global transactions, in which it has been participated, So it polls coordinators of global transactions, using this timeout as interval for resending request to coordinator.	+
`transmgr.-checkpoint_period`	int.	sec	0	Specifies time interval between two checkpoint. Checkpoint can be forced either by exceeding some limit value of transaction log size or after some specified period of time (if this timeout is non-zero).	+

Parameter	**Type**	**Unit**	**Default value**	**Meaning**	**Set**
`transmgr.-dynamic_reclustering_limit`	int.	bytes	0	Enable or disable dynamic reclustering of object. If dynamic reclustering of objects is enabled, all objects modified in transaction are sequentially written to new place in the storage (with the assumption that objects modified in one transaction will be also accessed together in future). This parameter specifies maximal size of object for dynamic reclustering. Zero value of this parameter disables reclustering.	+
`transmgr.log_file_name`	string	-	`"*.log"`	Name of transaction local log file	-
`transmgr.-history_file_name`	string	-	`"*.his"`	Name of global transaction history file	-
`objmgr.lock_timeout`	int.	sec	600	Deadlock detection timeout. If lock can't be granted within specified period of time, server consider that deadlock takes place and abort one or more client process to destroy deadlock.	+
`poolmgr.page_pool_size`	int.	pages	4096	Size of page cache. Increasing this value will improve performance of disk IO operations.	-
`poolmgr.data_file_name`	string	-	`"*.odb"`	Name of storage data file	-
`server.cluster_size`	int.	bytes	512	Maximal size of object cluster to be sent to client. Server can perform optimization of sending objects to clients. Instead of sending one object for each request, it can send several objects (cluster of objects), including in cluster objects referenced from requested object (but only if total size of objects will not exceed `cluster_size` parameter).	+

1. Last column of this table marks parameters, which values can be changed by set command in interactive mode.
2. Symbol `*` used in the file name in this table stands for concrete database name (parameter passed to GOODSRV).

Other arguments of `goodsrv` are:

```
goodsrv <storage name> [<storage-id> [<trace-file-name> | "-"]]
```

By default `goodsrv` starts in interactive dialogue mode, allowing user to execute some administrative operations with database as well as to see database usage parameters. Also if GOODS server was compiled with trace option. trace information will be outputted at the terminal. If you specify log file name as last argument to `goodsrv`, messages will be also saved in the specified file. Below is a list of all valid commands for `goodsrv` monitor:

```
help [COMMAND]                      print information about command(s)
open           open database
close           close database
exit           terminate server
log [LOG_FILE_NAME|"-"]             set log file name
show [CATEGORIES]                   show current server state
monitor PERIOD [CATEGORIES]         periodical monitoring of server state
backup FILE_NAME [TIME [LOG_SIZE]]  schedule online backup process
stop backup          stop backup process
restore BACKUP_FILE_NAME            restore database from the backup
trace [TRACE_MESSAGE_CLASS]         select trace messages to be printed
set PARAMETER=INTEGER_VALUE         set server parameter value
```

`CATEGORIES` is some of servers clients memory transaction classes. By default all categories are shown.

`TIME` is interval of time in seconds and `LOG_SIZE` is transaction log size after reaching one of this parameters backup procedure will be started. Default value for this parameters are `0`, which means that backup will be started immediately.

`SHOW_OPTIONS` is a list of some of the following options “servers clients memory transaction classes. If you skip this parameter, all possible characteristics are displayed.

`PARAMETER` is one of valid server parameters (see table above). Execute help set command to obtain list of all valid parameter names.

`TRACE_MESSAGE_CLASS` is a space separated list of message classes. Only messages belonging to one of the specified classes will be outputed. The complete list of message classes can be obtained by help trace command.

3.3.3 Database browser

Database browser is a program which allows you to dump fields of objects in database. I want to make this program as simple as possible and also system independent. The primary idea of writing this program is to show how metainformation can be extracted from database. There are two versions of browser: console application and CGI version using WWW browser to navigate through objects.

Browser can dump values of fields of all objects (including classes) stored in database storages. Console version of database browser browser.cxx requires single command line argument, which specifies name of database configuration file. It is not necessary to specify all servers in this configuration file, you only have to describe that storages, objects from which you want to inspect. Certainly servers of these storages should be previously started before you run ”browser” application. To refer object you should specify storage and object identifier. It is possible to see list of available commands by typing ”help”.

CGI version of browser cgibrows.cxx provides much more user friendly interface. To be able to use this browser you need some WWW server running at the computer where database storage is located. For example it can be Microsoft Peer WebServer, included in NT-4.0 distribution, or Apache free WWW server. You have to do some preparation before you can use the browser. You should edit file browser.htm (name can be changed) to specify name of your host (localhost is possible) and path to cgibrows program (cgibrows.exe in Windows). This program can be placed either in some default directory for CGI scripts used by WWW server, or it is possible to register another directory in WWW server. It is preferable to have cgibrows program in the same directory as database, otherwise you should specify full absolute path to directory where database is located when opening browser. Be sure that user, under which name CGI script will be executed, have enough permissions to access files and ports at local computer (GUESS by default has no such permissions).

That is all with preparation of WWW browser for GOODS. Now you can open in your favorite WWW browser the page browser.htm and specify database name and storage number. Database name is the name of configuration file without extension. If configuration file and cgibrows program are located in different directories, you should specify absolute path to this file. By default storage with identifier 0 will be opened. To start browser press Open button. Browser will dump fields of root object. Browser outputs non NULL reference fields as pair: storageID:objectID. To navigate through objects, click on reference field. Do not forget about Back, Forward and Go buttons, which can help you in navigation. You can browse database from any computer which can access WWW server.

3.3.4 Running GOODS examples

You can try to play with some examples of GOODS application programs:

`guess.cxx`	— game "Guess an animal",
`unidb.cxx`	— "university" database,
`testblob.cxx`	— work with binary large object
`tstbtree.cxx`	— program for measuring of GOODS performance

Program "guess.cxx" is the simplest database application using optimistic approach for synchronization. This program creates binary tree with information about animals (or anything else which you have entered). To run this program you should first start database server by the following command:

```
> goodsrv guess
```

Then you can start any number of client's sessions by running program guess in separate windows. Optimistic approach in this application means that if while you are answering program questions and other user has performed update of the same branch of the tree then you have to start from the beginning.

Program `"unidb.cxx"` is a sample of GOODS application working with distributed database. Structure of this database is very simple. University database contains `B_tree` of students and `B_tree` of professors. Each student has topic of diploma work and is attached to one of professors (his tutor). Professor is an owner of unordered set of students (group) attached to him. Students can be added, removed or reattached from one professor to another. Professors also can be added and removed, but it is only possible to remove professor which is not someone's tutor. This application shows a simple menu allowing user to select action and also uses GOODS change notification mechanism to handle database modification done by other clients (each time student or professor is added or removed by some database client, total number of students and professors in the university shown at the top of menu is updated).

Objects in this application are distributed between to storages: student's storage (0) and professor's storage (1). All student objects (and objects referenced from them, such as `string` and `set_member` objects) are placed in this storage. All professor objects are placed in professor's storage. To run this application you need to start two servers:

```
terminal-1> goodsrv unidb 0
terminal-2> goodsrv unidb 1
```

Then you can start any number of clients by running program "unidb" without any arguments at different windows. You can see example of dealing with large multimedia objects in "testblob" application. Two classes from GOODS class library are used in this application: `blob` and `hash_table`. Class `blob` provides incremental loading of big binary objects in parallel with it's handling (for example you can unpack and transfer data from buffer to audio device while next part of object is loading from database). Also this application tests multitasking at client's site. Program "testblob" is very simple and have only few commands, allowing you to insert, extract and remove files in(from) database. Type `help` command to learn more about command syntax. To run this application you should activate database server by issuing command "goodsrv blob" and then start application itself. One of the most effective structure for spatial objects is R-tree (proposed by Guttman). GOODS library contains `R_tree` class which implements Guttman's quadratic split algorithm. For testing of this class implementation very simple model of spatial database is developed: `tstrtree`. All object in this program are placed in R-tree and H-tree (combination of B*-tree and hash table) and can be accessed either by name or by coordinates. Configuration file for this program is "rtree.cfg". So issue command "goodsrv rtree" to start server and command `tstrtree` to run test program.

Parallel processes	Alpha-Server 2100 2x250 Digital-Unix 4.0 portable	Alpha-Server 2100 2x250 Digital-Unix 4.0 pthreads	PowerPC 120 Mk-Linux portable	PPro-200 Linux portable	PPro-200 WinNT 4.0	SPARC-station-20 2x50 Solaris 2.5 pthreads	PPro-233 FreeBSD 3.0 portable	Ultra-Sparc 2x300 pthreads
1	239	227	121	73	57	349	117	130
2	226	187	124	95	47	339	116	119
4	221	112	130	66	30	178	65	70
8	233	114	156	67	37	188	68	71
16	240	124	247	68	44	209	68	74

Table 2: GOODS transaction performance test

3.3.5 Measuring GOODS performance

Program `tstbtree` simulates parallel work of several client with database. To run this program you should first start `goodsrv` server with the following parameters:

```
> goodsrv btree 0
```

and then run some amount of client applications by `"spawn"` utility, for example:

```
> spawn 32 8 tstbtree
```

This command will 32 times invoke `"tstbtree"` program and at most 8 instances of this program will be executed simultaneously. Each instance of `"tstbtree"` program inserts 100 records of size randomly distributed in range of 6..1030 bytes in one of 4 B-trees, then 10 times repeats a loop of searching each of this records in the B-tree and at the end removes all created records from the B-tree. So after the end of test there are should be no records in database and it is possible to rerun test once again. As far as not a performance of B-tree implementation itself but performance of GOODS server is measured, then it was decided to reduce size of B-tree page to 4 entries to increase number of objects participated in transactions.

It is interesting to investigate dependence between number of programs executed in parallel and total system performance. In theory the best result should be obtained when there are exactly 4 concurrent applications (mostly they all will work with different B-trees and no synchronization between them is necessary). If there are more than 4 applications running in parallel then a lot of notification messages used to synchronize caches of this applications will reduce total system performance. The following table contains results (elapsed time in seconds of test execution) for some systems:

Graphic representation of this results

The time of this test execution mostly depends on time of synchronous writes to transaction log. Improving of system performance in case of executing clients requests in parallel is mostly obtained by merging synchronous write requests. The following table shows average number of merged synchronous writes for different systems and number of clients running in parallel:

Graphic representation of this results

It is also interesting to measure database performance with transaction log synchronous writes option switched off. In this case mostly efficiency of GOODS server components implementation and their interaction is measured.

Graphicrepresentationofthisresults

There are also two programs not working with database but which can be used for testing and measuring performance of socket and multitasking libraries: `testsock.cxx` and `testtask.cxx`.

Test for socket is performed only with local clients (server and client are at the same computer). This test consists of two programs (client and server) which interact with each other in the same way as in normal GOODS applications. Client sends 1000000 requests to server, waiting for server response for each request. Each client request consists of two parts: header and body. So sending request to server require two socket write operation. This program mostly measures efficiency of socket library implementation at concrete system (`UNIX_DOMAIN` sockets in Unix), but in case of Windows-NT/95 performance of GOODS local sockets implementation is tested. Table below represents results for different systems (elapsed time in seconds of test execution):

Program verb+testtask+ measures performance of multitasking library. This test simply starts a number of tasks (threads) each of them performs the following loop: wait for signal, enter critical section

Parallel processes	AlphaServer 2100 2x250 DigitalUnix pthreads	PPro-200 WinNT
2	1.009	1.854
2	1.845	2.948
8	1.805	1.953
16	1.758	1.767

Table 3: Parallel transaction log writes

Parallel processes	AlphaServer 2100 2x250 DigitalUnix 4.0 portable	AlphaServer 2100 2x250 DigitalUnix 4.0 pthreads	PPro-200 Linux portable	PPro-200 WinNT 4.0	SPARCstation-20 2x50 Solaris 2.5 pthreads	PPro-233 FreeBSD 3.0 portable	UltraSparc 2x300 pthreads
1	28	47	19	22	102	29	40
2	23	30	21	13	87	26	27
4	21	30	17	17	90	21	27
8	24	41	21	21	137	21	31
16	39	56	24	33	174	24	37

Table 5: GOODS transaction performance test with asynchronous writes

AlphaServer 2100 2x250 DigitalUnix 4.0	PPro-200 WinNT 4.0 WinSockets	PPro-200 WinNT 4.0 GOODS local sockets	PPro-200 Linux	SPARCstation-20 2x50 Solaris 2.5	PPro-233 FreeBSD 3.0 portable
374	275	25	109	541	86

Table 7: Performance of socket library

AlphaServer 2100 2x250 DigitalUnix 4.0 portable	AlphaServer 2100 2x250 DigitalUnix 4.0 pthreads	PPro-200 WinNT 4.0	PPro-200 Linux portable	SPARCstation-20 2x50 Solaris 2.5 pthreads	PPro-233 FreeBSD 3.0 portable
3	29	11	2	65	2

Table 9: Performance of GOODS multitasking library

4 Installation of GOODS

4.1 Compilation of GOODS sources

GOODS is now running under Windows-NT/95 and various Unix dialects. All system specific code is encapsulated within few files and is accessed by abstract system independent interfaces: `task.h`, `sockio.h` and `file.h`. There are several system specific implementations for this interfaces. To achieve maximal performance advanced features of modern operating system are used, such as memory mapped files, gathered io (writev), threads. That can be a source of problems with porting GOODS to some old Unix dialects.

To build GOODS you should execute config script in source directory. You can specify name of your system or let configuration script tries to guess target system itself. Configuration s cript only copies one of system specific version of makefiles to the file "makefile". There is common makefile for all Unix systems "makefile.uni" containing targets and rules. All system specific makefiles only define some parameters (such as CC, CFLAGS...) and include `makefile.uni`. The main difference between systems (except name of C++ compiler and compiler flags) are with type of multitasking library used by GOODS (portable, implemented by `setjmp/longjmp` or based on Posix pthreads).

It is not necessary to run configuration scripts at Windows. I am using Microsoft Visual C++ 5.0 for compiling GOODS. Makefile for windows has name `makefile.mvc`. There is special `MAKE.BAT` file which invokes `NMAKE` and specifies the name of makefile.

After configuration just issue make command to build GOODS. The following targets will be build:

- Server library `libserver.a` (`server.lib` at Windows);
- Client library `libclient.a` (`client.lib` at Windows);
- Database storage server `goodsrv`;
- Database browser `browser`;
- Several tests and sample applications;

4.2 Description of GOODS sources

- async.cxx
  Asynchronous event manager for portable multitasking library. Task with priority 0 repeatedly calls Unix KBD¿select function to find out channels ready to input/output.
- async.h
  Specification of asynchronous event manager.
- browser.cxx Simple database browser. This application shows how metainformation can be extracted from database storage.
- cgibrows.cxx

  CGI version of database browser. Using this CGI program you can browse database objects from WWW browser if WWW server is installed at your computer.
- class.cxx

  Classes to support class information for client application at runtime. Methods of this classes are responsible for building class and field descriptors, conversion of object instances when from storage format to application representation and visa versa.
- class.h

  Definition of classes providing reflection property to client application.

- classmgr.cxx

  Storage class manager implementation.

- classmgr.h

  Interface for storage class manager.

- client.cxx

  Implementation of application independent client interface with storage.

- client.h

  Definition of database storage interface for clients.

- config.h

  Definition of some global types and constants used in GOODS.

- console.cxx

  Implementation of GOODS console interface. This methods perform input and output data from terminal.

- console.h

  Definition of GOODS console interface.

- convert.h

  Definition of functions for (un)packing atomic types from storage format (big endian, unaligned) to application representation.

- ctask.cxx

  Implementation of portable non-preemptive multitasking library, using setjmp/longjmp function for context switching.

- ctask.h

  Definition of classes used by portable non-preemptive multitasking library.

- database.cxx

  Application dependent part of client interface with database. This part of interface is responsible for synchronizing access to database, packing/unpacking objects, handling servers messages.

- database.h

  Definition of application dependent part of client interface with database.

- dbscls.cxx

  Implementation of application database classes: set, B-tree, hash table, blob, dynamic arrays and strings.

- dbscls.h

  Collection of application database classes.

- file.h

  Abstract file interface. This interface provides operating system independent methods for working with files.

- goods.h

  Main include file for GOODS client application.

- goodsrv.cxx

  Simple database storage server program. This storage server is powerful and flexible enough for been used in various applications.

- guess.cxx

  Sample GOODS application: game "Guess an animal".

- memmgr.cxx

  Implementation of GOODS storage server memory manager with distributed and incremental garbage collector.

- memmgr.h

  Abstract interface for server memory manager.

- mmapfile.h

  Interface for mapped on memory file.

- mop.cxx

  Implementation of basic metaobjects. Using this metaobject which cover most common database access patterns, you can derive your own metaobject satisfying requirements of concrete application.

- mop.h

  Metaobject protocol definition. This protocol is used in GOODS for controlling object access synchronization aspects, transactions, and object cache management.

- multfile.cxx

  Implementation of file consisting of several physical segment (operation system files). Such multifile makes it possible to overcome operating system limitation for maximal file size.

- multfile.h

  Definition of multifile class.

- object.cxx

  Implementation of object class - base class for all GOODS objects. Also object index used for indirect object access is implemented in this file.

- object.h

  Definition of object class - top class in object hierarchy.

- objmgr.cxx

  Implementation of GOODS storage server object access manager. This manager is responsible for handling object locks and notification of clients about object instance deterioration.

- objmgr.h

  Definition of storage server object access manager interface.

- osfile.h

  Definition of file class corresponding to operating system file. There are several implementation for this class for different platforms.

- poolmgr.cxx

  Implementation of GOODS storage server page pool manager. Page pool manager is responsible for efficient work with database files.

- poolmgr.h

  Definition of storage server page pool manager interface.

- protocol.cxx

  Implementation of client-server protocol methods.

- protocol.h

  Definition of protocol used for client-server and server-server communication.

- ptask.cxx

  Implementation of multitasking library using Posix threads.

- ptask.h

  Definition of classes used in multitasking library for Posix threads.

- refs.h

  Definition of smart pointers for GOODS C++ interface.

- rtree.h

  Definition of R-tree class (effective search structure for spatial objects)

- rtree.cxx

  Implementation of R-tree class

- server.cxx

  Database server methods implementation. This server is responsible for coordination of work of all storage managers and interaction with clients.

- server.h

  Definition of database storage server class.

- sockio.h

  Abstract socket interface. Socket is reliable bidirectional connection used for implementation of client-server and server-server communications.

- spawn.cxx

  Auxiliary utility for spawning sever parallel applications. This utility can be used for testing GOODS performance.

- stdinc.h

  List of include files common for all GOODS modules.

- stdtp.h

  Definition of standard types and including standard system headers.

- storage.h

  Definition of application independent client interface to GOODS storage.

- support.h

  Set of support classes and functions for GOODS modules: dynamic arrays, buffers, hash functions...

- task.h

  System independent multitasking interface. This header file contains definitions of following classes: task, mutex, semaphore, event, eventex, semaphorex.

- template.cxx

  Template for GOODS client application.

- testblob.cxx

  Test program for binary large objects. This program can be used as example for creating your own multimedia objects.

- testsock.cxx

  Test program for testing socket performance.

- testtask.cxx

  Test program for GOODS multitasking libraries.

- transmgr.cxx

  Implementation of GOODS storage server transaction manager using maintaining log file for providing transaction recoverability. This manager handles local and global (distributed) transaction.

- transmgr.h

  Definition of interface for storage server transaction manager.

- tstbtree.cxx

  Test program for B-tree implementation in GOODS. This program can also be used for measuring GOODS performance.

- tstrtree.cxx

  Test program for R-tree implementation in GOODS (very simple spatial database).

- unidb.cxx

  Example of GOODS application: "university database".

- unifile.cxx
  Implementation of `os_file` class for Unix. This implementation supports concurrent access to file by several tasks, merging of parallel synchronous write requests, alignment of synchronous writes to operating system file block boundary. Two last capabilities greatly increase performance of server by making process of transactions commit more efficient.

- unisock.cxx
  Implementation of socket class for Unix.

- unisock.h
  Definition of class representing Unix sockets.

- winfile.cxx
  Implementation of `os_file` class for Windows. Optimizations include merging of synchronous write into single request to operating system.

- w32sock.cxx
  Implementation of socket class for Windows. This class uses `WinSockets` library for remote connections and provides own efficient implementation for local (within one computer) connection (UNIX domain sockets analog). This local sockets implementation use cyclic buffers in shared memory and is more than 10 times faster than WinSockets.

- w32sock.h
  Definition of socket classes for Windows,

- wtask.cxx
  Implementation of multitasking library for Windows.
- wtask.h
  Definition of classes from multitasking library for Windows.

5 Desripttion of client-server protocol

6 Distribution of GOODS

I will provide e-mail support and help you with development of GOODS applications.