# Efficient Transparent Application Recovery In Client-Server Information Systems

David Lomet and Gerhard Weikum
Microsoft Research
E-Mail: lomet@microsoft.com, weikum@cs.uni-sb.de

## Abstract

Database systems recover persistent data, providing high database availability. However, database applications, typically residing on client or "middle-tier" application-server machines, may lose work because of a server failure. This prevents the masking of server failures from the human user and substantially degrades application availability. This paper aims to enable high application availability with an integrated method for database server recovery *and* transparent application recovery in a client-server system. The approach, based on application message logging, is similar to earlier work on distributed system fault tolerance. However, we exploit advanced database logging and recovery techniques and request/reply messaging properties to significantly improve efficiency. Forced log I/Os, frequently required by other methods, are usually avoided. Restart time, for both failed server and failed client, is reduced by checkpointing and log truncation. Our method ensures that a server can recover independently of clients. A client may reduce logging overhead in return for dependency on server availability during client restart.

## 1. Introduction

### 1.1 Problem Statement

Database systems support fault-tolerance and high availability by recovering quickly from system failures. However, recovery has been restricted to the database and has ignored applications interacting with the database at the time of failure. Dealing with database system failures at the application level is still tedious even if the application itself stays alive. The application process is also failure-prone and would exhibit improved availability were it recoverable. Finally, complicated forms of inconsistency may arise from both application and database server failing independently within a small time window.

Thus, developing failure-resilient database applications remains a black art unless one limits application structure to special programming models, like queued transactions supported by TP monitors [Gray93]. TP-monitor recovery is limited to undoing incomplete transactions and restoring the last committed state of message queues. Because the state of application processes is lost, applications must be "stateless" between transactions. However, the rich states of modern applications cannot be reduced to a queued message. Consider long interactive sessions with a repository tool [Bernstein97], authoring tools [Kaiser97], workflow systems [Georgakopoulos95], a CAD system that may involve long-running computations, or a "middle-tier" application service such as SAP R/3, which invokes database services and itself supports many clients. Obviously, users of these applications would benefit from increased failure resilience, to minimize the amount of lost human work and to provide high availability at the application level rather than only the database system level. Ideally, recovery techniques should mask failures completely and quickly bring *both* the application and its underlying databases back to the pre-failure state.

We present an approach to database and application recovery that is both efficient and masked from the human users. Specifically, we consider a client-server environment where multiple clients run one or more applications, and all applications interact with the same database server. Applications are assumed to be "piece-wise deterministic", i.e., potentially non-reproducible behavior is caused only by exchanging messages with the server or the external world, typically a human user but also perhaps automatic instruments (e.g., in an embedded control system). Failures can be server crashes, client crashes, or both. Failures are assumed to be "soft", i.e., not lose information stored on stable storage such as disks, and "fail-stop", i.e., not lead to corruption beyond the actual point of failure. This captures most real system failures.

Transparent application recovery requires ***careful logging of messages and database updates***. Efficient client application and database server recovery from failures requires thorough cost/benefit reasoning:

- Low logging cost limits overhead during normal operation. In particular, *forced* log writes to stable storage, i.e. where the write must complete before execution proceeds, should be minimized.
- Fast recovery after a failure, of both server and application, is critical for high availability. In particular, short outages may be unnoticed by the human user.
- Recovery of server and clients should be as independently as possible. Especially, it is unacceptable for server restart to depend on possibly slow or unavailable clients.


SIGMOD '98 Seattle, WA, USA

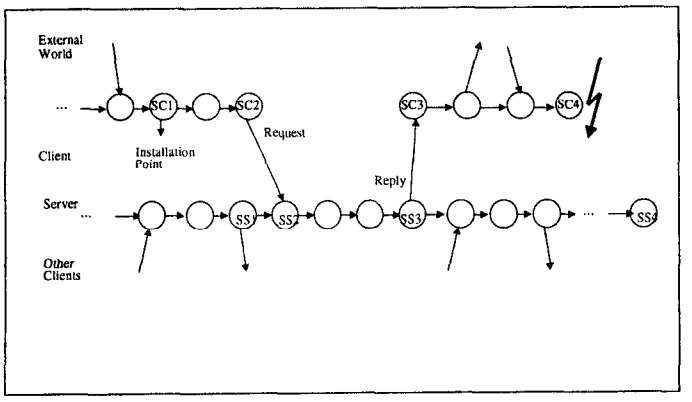


**Figure 1: Client-Recovery Problem Scenario**

These design goals require integration of recovery for client applications and the database server. Figure 1 shows a client application and a server proceeding through a series of states (shown as circles) with an exchange of messages at certain points. The server interacts with other applications as well. Suppose the client saves state SC1 of the application to disk. We call this an application installation point. Suppose this application fails in state SC4, while the server proceeds serving other clients and their applications. When the client restarts, it reincarnates the application at SC1, the most recent installation point. It then re-executes the application, re-sending the previously sent request to the server. If the server has not taken precautions, it will treat the request as a "new request", and send a new reply back to the client. However, the server is now in a more advanced state, say SS4, and has meanwhile executed other clients' requests. Therefore, the re-executed "new request" of the restarted client may no longer produce the same outcome as the original request and may show results to the user that differ from the original results. This problem would not arise if client failure and restart had forced the server to also restart from a state that has been synchronized with the client's restart point, for example state SS1. However, such an approach compromises server recovery independence and availability in an unacceptable way.

## 1.2 Prior Work

Prior work on application fault-tolerance in a distributed system involves application ***installation points*** and/or ***message logging***. Installation points are called "application checkpoints" in the literature, meaning that an application's state is saved to stable storage. We avoid this term to avoid confusion with database checkpoints, which are special log records permitting log truncation [Gray93, Mohan92]. The prior work can be categorized into three approaches, all of which incur high normal operation and/or recovery costs:

- ***Fault-tolerant process pairs***: This approach provides a *primary* process with a *hot-standby* process, usually on a different processor. When the primary process fails, the standby process takes over and re-executes starting from the last installation point of the primary process. The sending of regenerated messages is conditionally suppressed, based on testing sequence numbers against logged messages. This approach was pioneered in the early eighties [Bartlett81, Borr81, Borg83, Borg89, Kim84] but is a heavyweight solution justifiable only for mission-critical high-end applications. It requires either an installation point or a forced message log record at every process interaction. This frequently required disk I/O greatly limits the achievable throughput of both server and clients.
- ***Distributed state tracking***: Approaches from the distributed algorithms community (see, e.g., [Strom85, Johnson87, Strom88, Alvisi95, Elnozahy96]) have a relaxed model of communicating processes. Processes generate installation points only occasionally and independently. Messages are logged in an ***optimistic***, non-forced manner. A failed process restarts from its most recent installation point, but other processes may be forced to restart from earlier states to guarantee a causally consistent global state [Chandy85]. This induces recovery dependencies among the processes that are unacceptable for a database server. Further, the restored global state is not necessarily the last externally observed state. This is fine for "number-crunching" computations, an initial target of this work, but it does not mask application failures from the user. Thus, this work has had little impact on real systems [Huang95]. A variation of message logging that eliminates recovery dependencies is ***pessimistic*** message logging. Unfortunately, this approach is very expensive as it forces every log record immediately. Most of this research ignores the need both to minimize logging cost and to truncate the log for fast restart.
- ***Persistent queues:*** This approach requires interactions between processes to be via persistent queues [Bernstein90]. When a process sends a message to another process, the sender enqueues the message to a persistent queue within the boundaries of a distributed transaction involving the queue and the sender. This incurs the high forced logging costs of a ***two-phase commit*** protocol [Gray93]. Moreover, the same protocol is used when the receiver dequeues the message. This solution has been very successful in the context of transaction-structured applications such as reservation systems, and is even suitable for heterogeneous platforms. However, its disk I/O costs are very high, and each application must be decomposed into a sequence of transactions with no state external to the queued messages.

Our approach achieves essentially the same effect for client/server applications as the persistent-queue method or a pessimistic logging method, but with much lower logging costs and very fast restart. The key to this is to exploit the special properties of request/reply messaging, and to use advanced database-style logging and recovery techniques which minimize log forcing, batch log I/Os, and efficiently truncate the log. These techniques integrate application recovery with database recovery. The only prior integrated approach [Lomet97, Lomet98] carefully coordinates the writing of log records and the installing of updates for both database and application-state modifications. It handles applications that run under the direct control of the database system (resource manager) on the same machine. Client/server applications need new techniques.

## 1.3 Our Approach and Contribution

Our approach inherits the combined techniques of application installation points and message logging. In contrast to prior work, we take more care about when to force message log records versus when to batch them with other log records and write them lazily. Log records that capture input messages from the external world to a client application are usefully forced immediately. Opportunities for optimization do exist, however, for the log records that capture client-server ***request-reply*** interactions.

There are several options for distributing the logging work between client and server. Our method of choice, coined "***server reply logging***", makes the server responsible for the forced log I/Os. This is advantageous as the server can recover independently and the batching of log records is more likely at the server. A client can log server replies, too, for faster application restart, but it can do so lazily. Client recovery may then need the server to ship log records that the client has lost in a failure, but this is a small price for much better throughput. Besides, a client can limit its need for log record shipping. That the server keeps log records that may be necessary for the restart of a client application creates subtle problems for truncation of the server's stable log file. Log truncation is an important efficiency issue as it affects both log space consumption and the duration of the server's redo recovery pass and hence, server availability. We derive a low-overhead scheme that supports effective log truncation, based on the client notifying the server when it no longer needs reply log records for its recovery.

The paper improves the state of the art in three ways:

- Advanced database logging and recovery techniques are extended to integrate database server and client application recovery, while ensuring recovery independence for the server. The supported class of applications includes "stateful" client applications and is thus substantially more general than the "stateless" applications represented by sequences of transactions ("chained transactions") alone.
- Message logging techniques from the distributed algorithms literature are tailored to client-server application recovery to minimize logging I/O costs and effectively support log truncation for fast restart when the client/server interaction is request/reply. Compared to using transactional persistent queues between the client and the server, our method reduces the forced log I/Os by a factor of six.
- The paper reconciles concepts from separate research communities, database systems and distributed algorithms. The result is an efficient solution that makes failures transparent to users by providing high availability at both the database system and application level. We believe this is important for making application fault tolerance an affordable "commodity feature", not just an exotic "high-end luxury".

Throughout the paper we consider only redo recovery (equivalently, the "repeating-history" part of recovery [Gray93, Mohan92]) as the most important component of fault tolerance. We do not make any specific assumptions on the embedding of client requests into transactions. Multiple request/reply pairs are possible within a transaction, or a single request may comprise multiple ACID transactions (e.g., when the request spawns a stored procedure with multiple commit points). Also, our method does not rely on a particular isolation model of transactions; both full serializability and weaker isolation levels are compatible with the developed recovery algorithms.

Undo or compensation of certain application steps and/or database updates may also be necessary after a failure, but this is an orthogonal subject. Indeed, the capability for undo is needed during normal operation as well. This undo recovery is no different from what is already in the literature (see [Elmagarmid92, Ramamritham96] for overviews).

The rest of the paper is organized as follows. Section 2 introduces client-server information systems and their requirements for recovery, and discusses various design considerations. Section 3 presents the paper's core contribution, a detailed algorithm for efficient and user-masked application recovery, based on the design considerations of Section 2. Section 4 discusses additional refinements and optimizations of the algorithm, and an outlook on possible extensions, while Section 5 concludes the paper with a summary of the salient features of the algorithm. Pseudo-code for the developed algorithms and additional correctness reasoning can be found in [Lomet98a].

# 2 Design Rationale for Recoverable Client-Server Systems

We focus on request-reply interactions between client applications and a server. We assume that all requests of client applications are intercepted by the client's run-time system (e.g., the client's ODBC stub). We further assume that the server runs a single resource manager that can control both database updates and its incoming and outgoing messages in an integrated manner. The server does not depend on any application state information across multiple requests, other than what it keeps in its database. For example, when SQL cursor positions or temporary tables live across request/reply interactions, the server has to maintain them in its database so that they are recoverable. Finally, we assume that output parameters of request executions, including return codes, are part of the server's reply message.

## 2.1 General Message Passing

Consider two processes that exchange a message, a sender and a receiver. Either or both may fail. We must ensure that the recovered states of the two processes either both contain the message exchange or that neither does. This amounts to requiring the atomicity of a send-receive interaction and explains why the most successful prior approach has been based on persistent message queues (see Section 1.2). However, queues require two distributed transactions for each message exchange, and hence two instances of two phase commit. One transaction is between the sender and the queue manager (so that the sender's state is advanced if and only if the request is persistently enqueued), and a second between the receiver and the queue manager dequeuing the message. If the second transaction fails, the message is returned to

the queue, and the surrounding TP monitor guarantees that the receiver eventually retries the dequeuing. The net effect is that the state transitions of both the sender and the receiver form one atomic unit.

The drawback of the queue-based solution is its high I/O cost. Sender, receiver, and queue manager all force information to stable storage. Reducing this cost is one of the main points of this paper. As our algorithm demonstrates, it is sufficient that only the sender force-log the message. Then, if the receiver fails after having received the message but without making it stable, the receiver can again obtain the message from the sender, and this argument is essentially symmetric in the two roles of a process. In fact, this is obviously the weakest logging requirement that can satisfy the correctness criterion. However, this approach has certain non-obvious implications:

1. The sender takes the responsibility for recreating the message when the receiver fails and needs to replay the message exchange. The protocol also needs to consider when the sender can discard the logged message and hence the server needs to know when the receiver no longer has to re-obtain the logged message.
2. The receiver's recovery depends on the sender's having forced logged the message. Should the receiver fail after the message exchange, it cannot perform independent restart. It must communicate with the sender because the message exchange must be replayed, but the receiver itself may have no stable record for it.

Our goal is to improve upon sender logging for the important case of request/reply message pairs. We specialize in the rest of the paper to client-server information systems where we can make specific assumptions about the roles of the processes in order to design an efficient protocol.

## 2.2 Request/Reply Design Issues

Compared to general message-passing processes, the client-server scenario is special in a number of ways:

> ***Clients:*** A client knows exactly when one of its applications is waiting for a reply message and this application is suspended between sending a request to the server and receiving the reply. A client application interacts with only one other process, the server. A client application may interact with another application, either on the same or some other client, only via the server's shared database, which has its own recovery. A constant for all recovery scenarios is that the client application also can interact with a user, and hence has requirements imposed by that which we discuss below.
>
> ***Server:*** The server communicates with many clients concurrently. Hence, it can exploit batching to improve the disk I/O efficiency of logging. Furthermore, the server usually processes multiple requests of different clients concurrently. Since it does not have to commit itself to an ordering of these requests until it sends replies, it can perform optimizations that are impossible in a general message-passing framework. However, this means that the server will not be piece-wise deterministic between message events unless it does sufficient logging to be able to reconstruct the exact interleaving of database reads and writes. Such extensive logging can be expensive. Not interleaving request executions is unacceptable as it leads to poor server throughput.
>
> ***Client-Server Dependability:*** *The* server is much more reliable, because it is carefully administered, than the clients are. Therefore, client applications may be willing to rely on the server's availability, but the server should never depend on the clients – quite an asymmetric situation.

A decisive difference between client and server is that the client application is piece-wise deterministic between requests whereas the server is not.

## 2.3 Server Considerations

We could treat the server's concurrent request executions as a set of message-passing threads, whose "messages" correspond to the interleaved accesses to the shared database and each would have to be force-logged. Fortunately, the fact that the execution of interleaved requests is not exactly reproducible does not matter until the resulting effects propagate outside of the server, i.e., when a reply is sent to a client. Thus sending a reply ***commits the state of the server***. From this point on, the server promises that it will ***deterministically replay*** a previously executed request if a failed and restarted client should re-submit the request. This commitment has three aspects:

- *Recreate reply*: The reply for a re-submitted request must be identical to the original reply.
- *Redo database updates*: Effects of the original request on the server's database are redone if necessary and the re-execution of a request is idempotent.
- *Isolate other requests:* The redo of database updates does not alter the data values previously read by concurrently executed requests.

To illustrate why these commitments are necessary, consider the example in Figure 2. When the server fails and restarts after having executed the action sequence shown, it must recreate the original reply for request 2, and ensure that W2(y) is redone if necessary. If done by re-executing request 2, R2(x) must see the value previously written by request 1. This problem is orthogonal to transactions. A request-reply pair needn't coincide with the boundaries of a transaction. The request-reply pair may contain multiple transactions (e.g., an invocation of a multi-transaction stored database procedure) or be embedded in a transaction with multiple requests (a conversational transaction).

The server relies on the client to re-submit application requests should the client fail. The server guarantees to provide the reply as long as the it does not fail by keeping the reply for a completed request in a volatile data structure, so that it can be sent back when a request is re-submitted. The challenge arises when the server fails and then receives a re-submitted request after it restarts. Section 3 details how this is done.

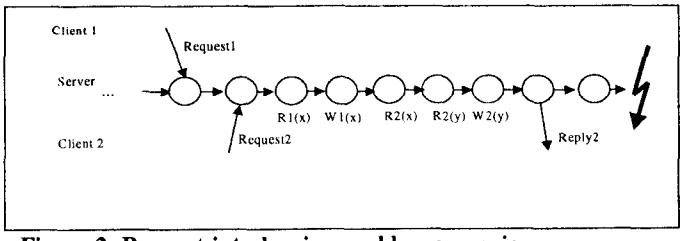


**Figure 2: Request-interleaving problem scenario.**

## 2.4 Client Considerations

Client applications, unlike the server, are piece-wise deterministic between requests. Hence, to recover the state of a client application, we need merely log the external input that it has seen and initiate replay from a saved installation point. The client exploits the recoverable request/reply mechanism when a request/reply needs to be replayed to recover one of its applications. The replay of the application up to the point of the request guarantees that the request is regenerated, the fundamental requirement placed on clients. A repeat of the request causes the re-delivery of the reply to the client application.

Application installation points are generated at the client, where the application is executing. The client may choose to store its installation points on a server for reliable storage, but the server then treats the installation-point information as regular data that it stores and retrieves upon the client's demand.

The client must also deal with input and output messages from and to the external world (e.g., the human user). It must log input messages and force them to stable storage promptly to minimize the frequency with which the user may have to re-submit input, so there is not much room for optimizations. Responsibility for logging these messages lies naturally with the client as the client receives them. For the rest of the paper, we assume that the client logs such external input messages. Note, however, that the input message log records should reside in the same log file that holds other client log records.

There is no need to log output messages to the external world, as they can be deterministically recreated if the application fails and restarts. During the restart, previously sent output is suppressed, except when an output message is immediately followed by a failure. Then, it is impossible to tell if the user has already received the output, whatever logging might be done. Therefore, such a message is re-sent and may thus be seen twice by the user. Note that should the "user" of our client be a program that can fail, then our "client" begins to look a bit like a server, and should start acting like one as well (see Section 4.3).

# 3 The Server Reply Logging Method

The above considerations strongly suggest that the server be responsible for the stable logging of reply messages. This enables fully independent server recovery after a failure, i.e., the server becomes available again without exchanging information with clients (that may be unavailable at this time). Server independence is a major design goal. Once we decide on server reply logging, most other aspects of the solution are natural consequences of our analysis. If replies are recoverable, requests are recoverable as well. Client applications are piece-wise deterministic, and hence their replay re-creates the request. So our server reply logging method need not perform any forced request logging on client or server. This leaves us with only one forced log I/O for each request-reply pair.

Server reply logging minimizes the server's forced logging frequency while providing the best solution in terms of recovery independence and fast server restart. The only aspect where it may be inferior to more client-centric approaches is a possibly increased restart time for client applications and that client recovery becomes dependent on the server. Given our design goals, especially server independence, and the fact the client depends on the server in any event, these disadvantages are surely acceptable. Therefore, our method of choice is server reply logging. In this section, we give a detailed description of the server reply logging method. We elaborate on the optimizations to minimize forcing the log; and on the subtle details of log truncation, on both server and clients, to speed up restart and, ultimately, guarantee very high availability. Pseudo-code for both client and server logging and restart is given in Appendix A.

## 3.1 High-level Description

The server forces reply records to the log before sending a reply. It does this by flushing the database log buffer, including the write log records of the corresponding request-reply pair, the reply log record being the last log record that must be written. In addition, the server may perform an optimization similar to group-commit [Gray93]; i.e., it postpones sending a reply until either a timeout occurs or a sufficiently large batch of log records has accumulated. It then writes the batch to the stable log file in a single disk I/O.

The server can choose among a number of options for replaying requests. It knows when a request execution is incomplete so that an undo scheme is applicable given appropriate isolation. Equally, it can choose replay a request to completion. Replay would now be necessary only for incomplete requests (i.e., when the reply is not yet stable). While these options require that the request and all database reads be logged in addition to the normally logged database writes, no additional ***forced*** log I/O is needed. Forcing the reply log record ensures that they are written to the stable log file as well. In fact, we will later see that there is an opportunity for further optimizations in this regard (see Section 3.3.1).

A failed and restarting client may re-submit requests, hence asking the server for some earlier replies. Because the server logs replies, the server is always able to look up the corresponding reply and send it back to the client without replaying its request. However, this may randomly access the server's stable log file, a potential disk I/O efficiency problem. Therefore, the server keeps the reply log records in a separate randomly accessible data structure, ideally in main memory, called the "***message lookup table***". During recovery after a server failure, the server rebuilds this table from its stable log to avoid random I/Os to the log for re-submitted requests.

From the client's viewpoint, a drawback of this method is that client restart time can be significantly longer than with client logging. Communication latency with a potentially highly loaded server to obtain the reply log records is a serious issue. To ease this problem, the client can perform some ***"lazy logging"***, writing reply log records to stable storage in a non-forced manner whenever there is available disk bandwidth.

Client recovery dependency on log records kept by the server causes subtle difficulties for ***log truncation*** at the server. Without additional mechanism, the server would never be able to truncate its stable log file and the log scan time during a restart would grow without bound. To avoid this, clients inform the server when log records are no longer needed by sending ***"stability notifications"*** to the server whenever the client:

- generates an installation point (log records preceding an installation point are not needed for client recovery); or
- makes reply log records stable by additional lazy logging of replies at low priority.

The client can piggyback stability notifications on its regular messages to the server. Application recovery is not compromised when the server garbage-collects reply log records that will never be used by a client.

## 3. 2 Data Structures

In addition to the usual recovery data structures, our method uses the following two data structures, instantiated at both the server and the client:

- an ***Active Application Table (AT)*** that contains status information about ongoing (possibly failed or restarting) applications that the server or the client is responsible for, and
- a ***Message Lookup Table (MT)*** that contains (log records about) messages of active applications, most importantly reply messages.

These data structures, described in Figure 3, reside in volatile storage, with entries made stable by forcing them to the log. We explain the various fields as we discuss the algorithm. A message is tagged with:

- an application identifier (AppID) that includes an encoding of the host client and is unique across all applications of all clients;
- a message sequence number (MSN) that is unique and monotonically increasing within each client application.

Messages include the input and output messages documenting client interactions with the external world. For convenience, we consider an application installation point (IP) as message with an MSN, and we distinguish start and termination installation points (start-IP and term-IP) from the regular ones.

The client tracks the last used MSN for each application, by recording it as the ***LastMSN*** in the active application table. The client keeps two additional MSN fields. The ***RedoMSN***, is the oldest MSN that follows the most recent installation point of the application, or the installation point MSN itself if no more recent message exists. The ***StableMSN***, is the most recent MSN for which it and all smaller MSN's of the client have stable log records. The client's StableMSN is the server's RedoMSN for the given application. All server message log records on its stable log file with an MSN smaller than the server RedoMSN of the corresponding application are obsolete and can be garbage-collected (they are unneeded by the client).

During the recovery analysis pass, relevant entries for the active application table and the message lookup table are recovered in volatile storage with information from the stable log file. Thus, server recovery has the usual two scans over the stable log file (analysis pass and redo pass), yet all relevant information on applications and reply messages is readily accessible for restarting clients.

```
AT: array[AppID] of record /* Active Application Table */
    LastMSN: integer;
        /* MSN of the application's last message event
        (only relevant on client) */
    StableMSN: integer;
          /* MSN such that all prior message events of the
          application, including this one, are known to be on
          stable storage locally (only relevant on client) */
    RedoMSN: integer;
        /* MSN of the oldest non-obsolete message event (for
        the client, this is always the most recent installation point
        or the oldest MSN that follows it) */
    RedoLSN: integer;
        /* LSN of log record that corresponds to the RedoMSN */
end;
MT: array[AppID,MSN] of record /* Message Lookup Table */
    MsgType: (request, reply, input, output);
        /* input and output are only needed for
        client-to-external-world messages */
    MsgContents: array of char;
end;
LF: persistent array[LSN] of record /* Stable Log File */
    LogRecType:  (write, read, undo, request, reply,
                  input, IP, start-IP, term-IP, CP);
    LogRecContents: array of char;
    AppID: integer;
    MSN: integer;
end;
```

**Figure 3: Major data structures of the server reply logging and recovery method**

## 3.3 Logging Algorithms

### 3.3.1 Server Algorithm

The server generates log records for each of its database write operations and each request and reply message, as well as some additional temporary log records to cope with incomplete requests, discussed below. These log records are posted in a conventional log buffer, which is forced to disk whenever it is full or according to write-ahead-logging or force-at-commit rules. In addition, the log records for messages are kept in the message lookup table described in the previous section. A reply log record is forced to stable storage, by flushing the database log buffer, before the reply message is sent to the client. Making the reply log

record stable does not imply that we discard it from the message lookup table. As a restarting client may re-request a log record, keeping the reply in main memory can save random disk I/O on the log file.

To force a reply message, the server flushes its log buffer in a single atomic write to the log. If the message lookup table still contains the corresponding request message (which must have the largest preceding MSN of the application), then this message can be discarded once the atomic log file write is completed.

*Logging for incomplete requests*

We need to take special care when the server fails in the middle of a request execution. Note that this problem arises with at most one request per active application, namely, the last, outstanding request of an application. The server has two principal options to achieve this goal:

1. *Rollback request, and execute again*: The server knows when the original ***reply has not yet been sent*** to the client. Then it has no obligation to deterministically replay as long as all database writes of ***concurrently executed requests are kept isolated.*** Thus the server can undo incomplete requests and re-execute them all as new requests with different interleaving and effects. This requires that the effects of incomplete requests be kept isolated. Isolation typically holds if a request-reply interaction is *inside one ACID transaction* that uses, e.g., strict two-phase locking. When a client request initiates a *sequence of transactions* on the server (e.g., a request that starts a "mini-batch" stored procedure on the server), undo requires compensating transactions. The effects of the original transactions would have to be kept quasi-isolated at a higher level of abstraction taking into account application semantics [Lomet92, Weikum90].
2. *Redo/continue request execution:* To make the server's non-deterministic behavior deterministically replayable, the server ***logs all database reads*** as well as writes. To ensure deterministic replay after the reply message is sent, the server must ***flush the log buffer before sending the reply***. This makes stable all read and write log records generated for the completed request execution. During server restart, a missing reply for an unfinished request is reconstructed by redoing and completing the request. During replay, database reads and writes are intercepted. Writes are applied to the database using LSN testing for idempotence. There are two ways in which to handle reads.
   a) Log ***physical read operations*** that record the values read by database reads. During recovery, read values are extracted from the logged values. The danger is that logging read values can greatly increase the amount of data logged.
   b) Log ***logical read operations*** that record the occurrence of a database read and its source [Lomet98]. During recovery, read values are re-read from the database. Thus, recovery must read the same versions as originally read. However, current cache managers write pages back to the stable database in any order, and hence may overwrite the version required by a read. Thus, the cache manager must not overwrite a page whose prior version is still needed to replay a read, by ***tracking and enforcing installation dependencies*** [Lomet95]. A danger is that installation dependencies may prevent the flushing of a dirty page (containing a version needed for recovery) for a very long time. Forced logging of the reply ends this installation dependency.

None of the options incurs additional forced log I/O for incomplete requests. Also, request log records and either undo log records or read log records are irrelevant once the reply log record is forced. Therefore, if any of these log records are still in the log buffer, they can be safely discarded without being written. This technique is similar to how atomic subtransactions are dealt with in the multi-level recovery methods of [Weikum90, Lomet92, Weikum93], where some log records become irrelevant when subtransactions commit. However, should these log records be written to the stable log, they must appear in the log in their chronological order for correct recovery.

There is no clear choice among our three options. For concreteness, pseudo-code in Appendix A is based on logging read values and redoing/continuing request executions (option 2a).

*Checkpoint log records*

As in standard database recovery, the server creates checkpoint (CP) log records that contain certain bookkeeping information to shorten the analysis and redo pass of recovery. The CP log records identify which parts of the log are irrelevant and effectively truncate the log. However, since we need to take into account both server and client needs, log truncation is more complicated than with database recovery alone. We postpone the detailed discussion of this until we have presented the client logging algorithm.

### 3.3.2 Client Algorithm

A client creates log records for each request and reply exchange with the server. In addition, it creates log records for each input message from the external world (human user or sensor/actor, e.g., in an embedded control system). These log records are kept in a message lookup table with the same layout as on the server. The client forces log records for external input to the stable log file immediately. The other messages are not forced to the stable log file. Rather the client writes them "lazily".

The client maintains a StableMSN for each application, which tracks by how much the client lags behind the server in terms of its stable message logging. The StableMSN is increased when the client writes a set of (chronologically) consecutive request and reply messages to the stable log file. One concrete policy is to ensure that this "backlog" (relative to the server) is limited by initiating a log file write whenever LastMSN – StableMSN reaches some threshold.

Each application periodically generates an installation point, saving the entire process state of an application onto stable storage on a per application basis, using a shadowing technique to provide atomicity of installation points. Each installation point is tagged with the MSN assigned to it and is thus "self-describing". Once an installation point is completed, all earlier log records of the

corresponding application can be discarded. They are now lower than the RedoMSN for the application, which is advanced to the MSN of the IP log record.

Like the server, the client can also occasionally write checkpoint log records to allow truncation of its log. Only the "discarded" log records above are truncated. The issue of log truncation is discussed in detail in the next subsection.

### 3.3.3 Log Truncation

Both server and client continuously truncate obsolete parts of both the stable log file and the message lookup table. This is important to free disk and memory space and shorten the log tail scanned during the restart redo pass. Log truncation, a form of garbage collection, is especially important for the server. If the server cannot delete log records after some time, then its log processing upon restart becomes excessively long, and the server's availability is compromised by its role in application recovery. Below, we first consider when clients can discard log records, which is the simpler of the two cases, and then discuss server log truncation.

***Client log truncation***

A client discards *all* log records and message lookup table entries of an active application at each new installation point for that application. However, when the client runs multiple applications simultaneously, this does not yet allow truncating the stable log file, as other applications may still need "old" parts of the log file. The client marks the progress for the installed application in the active application table by setting its RedoMSN entry to the MSN of the installation point. (For convenience, installation points are viewed as messages here, so that they can be identified by an MSN. We also tag installation points with the MSN of their associated log record.) The minimum RedoMSN among all active applications then determines the part of the log file that the client needs to keep. To reconstruct the minimum RedoMSN after a failure without having to scan the entire log, the client periodically generates a checkpoint log record that contains the active application table. This is a standard technique of database-style logging and recovery, applied here to application message logging. The applications play the role of dirty database pages.

***Server log file truncation based on client stability notifications***

As the server does not itself install applications, it has no *direct* information about when it can safely discard log records and thus truncate its log file. The server relies on the clients notifying it about their steps that allow it to discard log records. These stability notifications need not incur extra messages as the relevant information is piggybacked on the next request message. The client steps that trigger a stability notification are an application installation or the writing of reply log records to the client stable log file. In both cases, the client increases its StableMSN, and it is this that is sent to the server. Upon receiving a stability notification with StableMSN m, the server discards all entries of its message lookup table with MSN's smaller than or equal to m. The server also sets the application's RedoMSN in the active application table to the smallest MSN higher than m and adjusts the corresponding system RedoLSN accordingly.

The server needn't make the changes to its message lookup table and active application table stable. It should, however, generate a log record to mark this event in the log, but this (very short) log record need not be forced. Consequently, when the server fails, it may not remember that it effectively truncated its log file and it may scan the log starting from overly old RedoMSN's. This does not affect correctness and the stale RedoMSN information will usually be updated soon by the next stability notification from the client.

The only case where a stale RedoMSN could result in a permanent and critical problem is the server loss of a stability notification from a terminating application that will not send more notifications. Log records for this application would become permanently unreclaimed garbage, forever preventing the server from truncating its log. Thus, we require that stability notifications for application termination be specially tagged. The server must force a "term-IP" log record to the log or generate a checkpoint record with an updated copy of the entire active application table. The client must await an acknowledgment from the server before it can remove the application from its own active application table and thus commit the termination.

## 3.4 Restart Algorithms

### 3.4.1 Server Algorithm

After a server failure, the server restarts by performing an analysis pass and a redo pass over the stable log file. The analysis pass starts from the most recent checkpoint log record (found by looking up the bootstrap file) and scans all log records until the end of the log. For application recovery, this pass rebuilds the active application table. This table is re-initialized from the checkpoint log record, and is then updated whenever the log scan encounters an installation point log record for an application. At the end of the analysis pass, the server knows for which applications it may have to recreate replies. It also knows a RedoMSN and a corresponding RedoLSN for each application, bounding the part of the log that contains the required reply log records.

The redo pass then starts from the minimum of RedoMSN's among active applications or the minimum RedoLSN among dirty database pages, whichever is older. For client application recovery, we focus on the redo of the message log records, understanding that request execution can write to pages of the database, and hence that we must, in this redo scan, do normal database redo as well. The server redo pass rebuilds the message lookup table, restoring it to its state as of the crash. The server can then deliver logged replies to re-submitted requests in case a client application has failed and is itself restarting.

A case that needs special consideration is the handling of incomplete request executions where the server has logged redo steps for database writes of a request. Then the request log record itself and corresponding undo or read log records are guaranteed to be on the stable log file at their original points in the interleaved request-execution history. We consider two cases.

- Undo log records permit the server to undo the database writes of all incomplete requests, subsequently re-executing these requests as if they were new requests, and re-generating the corresponding replies (option 1. in Section 3.3.1, valid for isolated request-reply interactions only.).
- Read log records permit the server to deterministically replay the partial execution of the incomplete requests and then continue executing the requests to completion (options 2. and 3. of Section 3.3.1).

The replies are then handled like replies during normal operation: they are inserted into the message lookup table, forced to the stable log file, and finally sent to the client. From this point on the server is in its normal operation mode. Client requests that are lost because the request log record is not forced are re-executed when the client re-submits the request.

### 3.4.2 Client Algorithm

Client restart also consists of analysis and redo passes over its stable log file. The analysis pass is identical to the server's and rebuilds the active application table. The redo pass, however, differs from the server's. The server "merely" rebuilds bookkeeping data that may be needed by failed clients. Client recovery actually restarts the applications active at the client failure, resuming their execution in a way that is transparent to the human user. This leads to the following differences.

The RedoMSN of an application identifies the oldest log record needed for recovery, the installation point log record, or the application log record following it. Note that the RedoMSN determined by the analysis pass is a lower bound on the real RedoMSN. It is possible for an installation point to occur just before the system fails and for the log record describing it not to reach the stable log. In this case, a better RedoMSN can be determined from reading the application installation point and examining its tag MSN.

The redo pass scans the log from the oldest RedoMSN of all applications in the reconstructed active application table. We process the log records of all applications at this client in a single pass over the log as opposed to making a separate pass per application. This is an important optimization for clients running "middle-tier" application services with a large number of concurrently active applications. Each application is re-incarnated upon encountering its analysis-determined RedoMSN log record (i.e., the lower bound for the RedoMSN) during the redo pass. The MSN that tags the application installation point is then used to determine the true RedoMSN, which may be later because the system may have failed before the stable writing of the log record for this installation point. All log records for the application are ignored until the true RedoMSN is encountered. This is the application analog to the way that updates to data pages are bypassed when the log record LSN for the update is less than the LSN stored on the page.

An application then re-executes asynchronously to the further redo processing of the log (i.e., in a separate process or thread). As in normal operation, application requests are intercepted. At these points, either client recovery has already encountered the corresponding reply log record or the application process pauses until it is encountered. If the reply has been encountered, then it is replayed. Otherwise the application waits and replays the reply when it shows up on the log. Application replay proceeds after the reply has been redone. Note that with asynchronous application re-execution, restart is substantially faster, which is especially important if applications perform long computations between server interactions.

The redo pass over the log file proceeds in parallel with application re-execution. It re-creates the message lookup table to the state as of the last stable log record. During the application re-execution, user input messages are consumed from the message lookup table, and output messages to the user can be re-created as part of the application re-execution. All output messages which are known to be followed by a log record of that application are suppressed (i.e., not sent to the user) as they would be duplicated.

There is a chance of repeating the very last output message or of missing the very last input message. Regardless of how quickly we force the log, a system failure can occur between the time of the input and the time when the input is logged. The best that we can do is to reduce the probability that this will occur. Similarly, on output, it is possible for the output to be lost before the user sees it, even if it were "sent". Regardless of logging, it is impossible to tell whether the user has seen the output or not until the user "acks" the message in some way. Thus, one has to re-present the last (un-ack'd) output.

For requests and replies, the client's analysis pass also reconstructs the StableMSN for each active application. So the redo pass knows which log records it will eventually encounter on its stable log file. All other, more recent reply log records have to be retrieved from the server. However, the client does not know its exact point of failure. Therefore, it does not know if there are additional log records on the server. When application re-execution reaches the last locally logged reply, application execution simply continues beyond this point and re-enters normal operation. The client executes the application until the next interception point and sends the request to the server. The client cannot (and does not have to) tell whether this request is a re-send or if it is the original send. On a re-send, the server sends back a previously logged reply. For a new request, the server does its normal request execution. An obvious optimization is to ask the server, right after the client's analysis pass, to asynchronously ship the reply log records that are more recent than the client's StableMSN. This approach is more complicated and therefore not pursued here.

# 4 Additional Considerations

## 4.1 Further Optimizations

Here we briefly discuss further potential improvements for handling the logging required for application recovery.

- ***Server log truncation***: When the RedoMSN of an active application becomes too old, it prevents log file truncation, hence increasing recovery time. To deal with this without client help, the server appends again the log record identified

by RedoMSN to the stable log tail. This changes the RedoLSN of the application, but not the RedoMSN. It allows the server to advance the redo scan start point and truncate the log. The server can iterate this technique, garbage-collecting interspersed obsolete log records by copying the "live" ones to the log tail. The message log records of an application now no longer appear in MSN order on the stable log file, but this is not a problem. At restart, the redo pass entirely rebuilds (the non-obsolete part of) the message lookup table anyway before resuming normal operation.

- ***Separately storing large replies***: We expect most request and reply messages to be short, but some applications may use very large parameter values in their messages, for example, when dealing with images or large documents. Such large log records increase the log processing time during a server restart, even when they are already obsolete but not yet truncated. Thus, it can be beneficial to separate message contents from the message log record. Such a log record contains only a pointer to a file (or disk address) where the message body is kept as an "extension". Finally, these extensions can be garbage-collected earlier because they are not interspersed with other log records.
- ***Regenerating replies via replay***: One way to avoid large reply log records is not to log message replies. If the total size of the request log record and all its read log records is significantly shorter than the reply log record, the server can choose to log that information instead of the reply. We preserve the server reply logging advantage of a single forced log I/O per request-reply. One can think of this as a way of "compressing" the reply message log record.
- ***Out-of-order client message logging***: The client can reduce its logging costs at the cost of complicating the server's log truncation by relaxing the order in which it writes log records. For example, the client may avoid writing large log records to its stable log file, instead writing shorter but more recent log records. Since the client's logging is "lazy", the large log record might never be made stable since an application installation might make this unnecessary. However, "out-of-order" log writing complicates both server log file truncation and client restart. Client stability notifications to the server must now contain explicit MSN's of stable log records rather than merely its StableMSN. The server is potentially more restricted in truncating the log. Client restart has to identify replies from the server explicitly rather than with a "high-water mark".
- ***Strenuous measures for difficult clients:*** The server must keep all "un-ack'd" reply log records. When a client does not send any stability notifications for an extended time, the server is sorely "inconvenienced". The server can ease this inconvenience with the techniques for server log truncation and the storing of large replies discussed above. Alternatively, at the cost of no longer masking an application failure, the server can discard such an application's log records. If the client fails and application restart takes unusually long, when application restart eventually asks the server for a reply log record, the server sends back a return code indicating a "cancellation" of that application. This does not do much harm if it is sufficiently infrequent. This seems appropriate, as a long application outage is no longer masked from the user.

## 4.2 State Installation

We have implicitly assumed that application state can be captured and installed at the client. This is essentially the stateless server model, i.e., the server holds no state for the client. However, database systems are not stateless servers. Clients have sessions (connections) with databases; sessions may have temporary tables, cursors, and a variety of additional state. There is nothing *conceptually* difficult in dealing with this. However, there can be substantial practical difficulties.

There are two generic approaches.

- The server permits the client to "re-install" server state when client redo recovery needs to instantiate the application state. Such server state includes, e.g., session-related information like session id, which are required if the server is to preserve any part of this state for the client.
- The server recovers these elements of client application state during its recovery. These elements are thus present for the client as part of the server state. Things like temporary tables or cursors are very difficult for the client to log or include in an application's installation point.

Further, we have assumed that request executions are independent and short enough so that we do not have to deal with installing intermediate states at the server. If this should be a problem, then we would need installation points for request executions as well as client applications. Such installation points are implemented using the same techniques as for client applications.

## 4.3 Extensions to Other Architectures

We have focused on an architecture where multiple clients interact with one server. It is straightforward, however, to generalize our approach to multiple servers. Each server simply employs the server reply logging algorithm, and on the client side, the data structures of the algorithm have to be instantiated and maintained for each server connection. No other extensions are needed.

Three-tier systems have a middle-tier service such as SAP R/3 with multiple client applications that interact with each other as well as with servers. Even more general architectures are possible with workflow management. A workflow engine is a middle-tier service that uses a database server to store the workflow state. It also interacts with "invoked applications" that run on different servers and in turn interact with other database servers or even other workflow engines. This leads to servers that may also be clients of other servers. As long as the client side of such a "mixed" architecture isolates all its execution threads and the threads do not issue asynchronous requests to other servers, our approach generalizes in a straightforward manner. Essentially each "mixed" application executes both the client and the server logging and recovery algorithms. However, if such a process has itself shared state among its concurrent executions or arbitrarily interleaves its executions with its requests to other servers, then our original assumption of piece-wise deterministic clients no longer holds. Providing transparent application recovery for this general architectural setting requires more research. We consider this an important direction for computer science.

## 5 Summary

We have analyzed the problem of making transparent application recovery efficient. Our algorithm for a client/server architecture, server reply logging, has a number of unique and novel properties:

- It substantially reduces the log I/O costs compared to conventional approaches based on persistent queues or message logging. It requires only one forced log I/O by the server before a reply is sent. Contrast this with the forced I/Os in the queuing approach, two each in three distributed transactions.
- It allows the server to recover and resume normal work independently of the clients.
- It ensures fast server restart consistent with modern database recovery techniques that perform a single redo pass over the log file (in addition to a short analysis pass). In addition, it facilitates continuous log truncation to bound the restart time.
- Although independence and restart performance of client application recovery is less important than for server recovery, the algorithm permits the client to make trade-offs. It supports fast application restart by allowing the client to "eagerly" log work to reduce its dependency on the server during a restart.
- The algorithm can be easily combined with a standby-process approach where a backup process is initialized and advanced using the logged messages to eliminate rebooting and process launching delays during restart. The backup process is typically run on a different processor, cluster node, or other unit of fault containment, and the failover is accomplished by shipping log records.

Our approach provides transparent application recovery. By its fast client restart, it can mask application failures to the human user. It is the first method that ensures an exactly-once semantics for applications as an affordable "commodity-feature". Transactions, on the other hand, provide atomicity in the "all-or-nothing" sense, where the "nothing" case requires explicit programming to re-initiate an application. Our approach allows none, one, or multiple transactions inside an application. Thus, it is more general than purely transactional applications. We believe that we have accomplished an important step towards providing "TP-monitor-class" system guarantees to arbitrary client applications.

## References


[Alvisi95] Lorenzo Alvisi, Keith Marzullo: Message Logging: Pessimistic, Optimistic, and Causal. International Conference on Distributed Computing Systems, 1995

[Bartlett81] J.F. Bartlett: A NonStop Kernel, ACM Symposium on Operation Systems Principles, 1981

[Bernstein97] Philip A. Bernstein, Brian Harry, Paul Sanders, David Shutt, Jason Zander: The Microsoft Repository. Invited Keynote Paper, VLDB Conference, Athens, 1997

[Bernstein90] Philip A. Bernstein, Meichun Hsu, Bruce Mann: Implementing Recoverable Requests Using Queues, ACM SIGMOD Conference, 1990

[Borg83] Anita Borg, Jim Baumbach, Sam Glazer: A Message System Supporting Fault Tolerance. ACM Symposium on Operating Systems Principles, 1983

[Borg89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, Wolfgang Oberle: Fault Tolerance Under UNIX, ACM Transactions on Computer Systems Vol.7 No.1, February 1989, pp. 1-24

[Borr81] Andrea Borr: Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing. VLDB Conference, Cannes, 1981

[Bulterman95] Dick C.A. Bultermann and Lynda Hardman: Multimedia Authoring Tools: State of the Art and Research Challenges, in: Jan van Leeuwen (Editor), Computer Science Today: Recent Trend and Developments, Springer, LNCS 1000, 1995

[Chandy85] K.M. Chandy and Leslie Lamport: Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computing Systems Vol.3 No.1, Feb. 1985, pp. 63-75

[Elmagarmid92] Ahmed K. Elmagarmid (Editor): Database Transaction Models for Advanced Applications. Morgan Kaufmann, 1992

[Elnozahy96] E.N. Elnozahy, D.B. Johnson, Y.M. Wang: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report, Carnegie-Mellon University, Pittsburgh, 1996

[Georgakopoulos95] Dimitrios Georgakopoulos, Mark Hornick, Amit Sheth: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases Vol.3 No.2, 1995, pp. 119-153

[Gray93] Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993

[Huang95] Yennun Huang, Yi-Min Wang: Why Optimistic Message Logging Has Not Been Used In Telecommunications Systems. International Symposium on Fault-Tolerant Computing Systems, 1995

[Johnson87] David B. Johnson, Willy Zwaenepoel: Sender-based Message Logging. International Symposium on Fault-Tolerant Computing Systems, 1987

[Kaiser97] Gail E. Kaiser and Jim Whitehead, Collaborative Work: Distributed Authoring and Versioning. IEEE Internet Computing Vol.1 No.2, 1997, pp. 76-77

[Kim84] Won Kim: Highly Available Systems for Database Applications. ACM Computing Surveys Vol.16 No.1, 1984, pp. 71-98

[Lomet92] David Lomet: MLR: A Recovery Method for Multi-Level Systems. ACM SIGMOD Conference, 1992

[Lomet95] David Lomet, Mark Tuttle: Redo Recovery after System Crashes. VLDB Conference, Zurich, 1995

[Lomet97] David Lomet: Application Recovery with Logical Write Operations. Technical Report, Microsoft Research, Redmond, Washington, June 1997

[Lomet98] David Lomet: Persistent Applications Using Generalized Redo Recovery. IEEE Int. Conference on Data Engineering, Orlando, FL 1998

[Lomet98a] David Lomet, Gerhard Weikum: Efficient Transparent Application Recovery in Cient-Server Information Systems, Technical Report, Microsoft Research, Redmond, Washington, 1998

[Mohan92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, Peter Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Transactions on Database Systems Vol.17 No.1, March 1992, pp. 94-162

[Mohan93] C. Mohan: A Cost-effective Method for Providing Improved Data Availability During DBMS Restart Recovery After a Failure. VLDB Conference, 1993

[Ramamritham96] Krithi Ramamritham and Panos Chrysanthis: Advances in Concurrency Control and Transaction Processing. IEEE Computer Society Press, 1996

[Strom85] Robert E. Strom, Shaula Yemini: Optimistic Recovery in Distributed Systems. ACM Transactions on Computer Systems Vol.3 No.3, August 1985, pp. 204-226

[Strom88] Robert E. Strom, David F. Bacon, Shaula A. Yemini: Volatile Logging in n-Fault-Tolerant Distributed Systems. International Symposium on Fault-Tolerant Computing, Tokyo, 1988

[Weikum90] Gerhard Weikum, Christof Hasse, Peter Broessler, Peter Muth: Multi-Level Recovery. ACM PODS Symposium, Nashville, 1990

[Weikum93] Gerhard Weikum, Christof Hasse: Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism. VLDB Journal Vol.2 No.4, 1993