

Transactions

Characterized by the following ACID properties.

- **Atomicity:** either all the effects of a transaction appear or none do.
- **Consistency:** a transaction transforms a consistent system state into another consistent system state.
- **Isolation:** Effects of a transaction are hidden from other concurrently executing transactions.
- **Durability:** Once the transaction executes successfully, its changes to the state survive all subsequent malfunctions.
- Consistency and isolation — concurrency control.
- Next few classes: ensuring atomicity and durability of transactions.
- If there are no failures and transaction programs do not abort atomicity and durability are trivially ensured.

Failures

- Transaction Failures: integrity constraint violation, process failure, deadlock resolution, scheduler initiated abort, . . .
- System Failures: power outage, OS bug, congestion and system overload, installation of new software, . . .
- Media Failures: disk arm crash, corruption of data on disk, . . .
- Natural disasters: earthquake, fire, floods, etc.
- Site failures in a distributed system.
- Communication Failures: message loss, link failure.

Recovery

- Recovery algorithms take the system into a consistent state from failure.
- Transaction Abort: remove the effects of the transaction from the database.
- System Restart: abort all transactions active at time of system failure, reinstate the effects of all committed transactions.
- Media Restart: abort all active transactions at time of media failure, recreate the database from the archival copy.
- Backup Takeover: in a disaster recovery system.

Recovery Algorithm Design Criteria

The design must consider:

- Frequency of failures (MTTF).
- Efficiency of the recovery algorithm (MTTR).
- Overhead on normal processing.

Frequent failures (low MTTF) \Rightarrow efficient recovery (low MTTR) + high overhead will be tolerated.

Infrequent failures \Rightarrow high MTTR tolerated + should not cause high overhead to normal processing.

Transaction failures are common (5 - 10 percent of transactions may abort.

 abort must take the same order of magnitude as the forward transaction.

System failures are rare.

 Depending upon application, 2 to 5 minutes for restart may be acceptable.

disaster recovery is extremely rare

 only a few databases containing sensitive data currently support this currently.

Recovery Algorithms Constraints

- correctness requirement: *recoverability* of schedules.
- efficiency requirement: must be able to remove partial effects of transactions by undoing the effects of the transaction in reverse chronological order using the logs (this requires strictness).

Recovery Algorithm Basics based on UNDO/REDO logging

- Transactions write UNDO and REDO log records during normal DO processing.
- The concurrency control manager enforces that schedules are strict.
- The system enforces write ahead logging and force log records at commit.
- Periodically, the system executes a checkpoint.
- If a transaction requests an ABORT_WORK, the UNDO log records used to remove the effects of the transaction. UNDO is done in reverse chronological order (note importance of strictness for the correctness of rollback).
- If a system fails, then on recovery, a system RESTART is executed.
- During RESTART, first the system finds out which transactions had committed and which had not committed before failure.
- REDO logs used to make effects of committed transactions persistent. (notice the requirement for force log at commit, and the importance of checkpointing to limit the amount of work in redo processing).

Recovery Algorithm Basics based on UNDO/REDO logging

- UNDO log records used to remove the effects of the transactions that did not commit before failure. (notice the importance of write ahead logging).
- Notice that the restart algorithm must be idempotent since a failure can occur while RESTART executes.
- Based on the above ideas numerous recovery algorithms can be designed. Important considerations in the design of the recovery algorithm are:
 - What actions are taken during checkpointing.
 - What kind of log records are written.
 - What granularity locks are used by the concurrency control mechanism.
- Recovery algorithms are very tricky– lot of incorrect and/or partial solutions have been suggested in the literature.
- Good papers to read: ARIES by Mohan et. al., MLR by Lomet, and Gray and Reuter Chapters 9, 10, and 11.

Logging Strategies

- *Value logging or physical logging*: store before and after image of objects. Alternatively, if the object is large (and changes small) store the changes to the object.
- *Logical Logging or operation logging*: store the name of an UNDO-REDO function and its parameters rather than object values.
 - Example: insert record r_1 into table T_1 .
 - Such an insert may result in insertions into multiple indices, may cause structure modifications to the index structures (page-split in B-trees), cause disk space to be allocated for a new file extent, etc.
 - Thus, a single logical log record corresponds to multiple (tens to hundreds) of physical log records.
- *Physiological Logging or page level operation logging*: Physical to a page, but logical within a page. That is, logging of operations on a page basis is done.
 - Example: consider an insert operation into a table T_1 with indices I_1 and I_2 .
 - Assume the operation results in updates to pages p_1 belonging to T_1 and pages p_2, p_3, p_4 belonging to I_1 and p_5 belonging to I_2 .

- Unlike logical logging, a log record will be written corresponding to the operation on each page.
- Which strategy is the best?

Undoing an Operation using UNDO logs

- **Requirement:**

$$\{x_{old}\}DO\{x_{new}\} \Rightarrow \{x_{new}\}UNDO\{x_{old}\}$$

- **Conditions under which the above holds:**

- Physical Logging:

- * UNDO operation corresponds to reinstalling before image of object using the UNDO log.
 - * UNDO will work correctly as long as the locks acquired for DO cover the UNDO operation.
 - Locks cover UNDO if page level locking used.
 - Locks cover UNDO if record level locking used as long as records are of fixed size.
 - * Note that UNDO operation is idempotent– it does not matter how many times we undo!

- Physiological Logging:

- * UNDO operation corresponds to the inverse operation of the original DO operation on the page.
 - * UNDO will work correctly as long as locks cover the UNDO operation and the record updated on the page does not migrate to some other page (else, undo will not be possible).
 - * Record level locking is sufficient.

- * Records can be prevented from migrating by making any operation that results in migration of the record (e.g., restructuring operations) conflict with the original operation.
- * UNDO operations may not be idempotent!
- Logical Logging:
 - * UNDO operation corresponds to the inverse operation of the original DO operation. Like the DO operation, the UNDO may cause changes to multiple pages.
 - * UNDO can be performed as long as enough locks are obtained during the DO part to guarantee success of the UNDO operation.
 - * Notice that restructuring or moving of records between pages does not create a problem for UNDO.
 - * UNDO and REDO operations may NOT be idempotent — inserting more than once may result in 2 records being inserted– if for example the index was not unique!

Comparison of Logging Strategies

- **Volume of Log Data:**

- Logical logging produces much lesser log records compared to physiological or physical logging. 1 logical log record may correspond to 10-1000 physiological/physical log records.
- Physiological log records may be smaller than physical log records.
- So logical better than physiological which is better than physical.

- **Concurrency Supported:**

- Physical logging forces page-level locks unless records are of fixed size (in which case record-level locking can be supported).

Consider modification by transaction T_1 of a record r_1 on page p_1 which reduces size of r_1 . If record locking used and freed space allocated to some other transaction T_2 which commits, then abort of T_1 will cause undo of the change made to r_1 by T_1 causing loss of consistency.

- Physiological logging will work with record locking as long as records do not migrate to different pages.

Consider a leaf level B-tree page p_1 containing records r_1, r_2, \dots, r_6 . Assume that transaction

T_1 inserts r_7 in p_1 . Now consider T_2 which tries to insert r_8 in p_1 but the page fills up causing a split into p_1 and p_2 . After split p_1 contains r_1, r_2, r_3 and r_4 , and p_2 contains r_5, r_6, r_7, r_8 . Now T_2 commits, but T_1 aborts. So the abort processing of T_1 will try to undo the insert of r_7 in page p_1 – but r_7 is no longer in p_1 !

- Records can be prevented from migrating by making any operation that results in migration of the record (e.g., restructuring operations) conflict with the original operation.
- Logical Logging: record level locking can be used and records are free to migrate to different pages.
- Logical logging permits highest concurrency, followed by physiological, and physical logging supports least concurrency.

Logical Logging seems to be the best solution!

Operation Consistency

- Notice that we have so far assumed that the execution of each operation, by itself, is atomic— that is, each operation either completely executes or does not execute at all.
- This property is referred to as **operation consistency** by Lomet in SIGMOD 92 and as **action consistency** by GRAY in Chapter 10.
- Operation consistency is important for both UNDO and REDO operations to succeed.
 - Correct UNDO requires that results of the original actions all be present.
 - Correct REDO requires that no results of the original action be present.
- Unfortunately, action consistency could be violated since failures may occur during the execution of the operation.
 - Logical Failures: violation of a consistency constraint.
 - Limit Failure: file too long.
 - Contention Failure: deadlock resolution.
 - Media Failure: damaged page cannot be written.
 - System Failure: system crash during action.

- If the UNDO and REDO satisfied the following requirements, then violation of operation consistency will not pose a problem.

$$\{x_{old}\}DO'\{x'_{new}\} \Rightarrow \{x'_{new}\}UNDO\{x_{old}\}$$

$$\{x_{old}\}DO'\{x'_{new}\} \Rightarrow \{x'_{new}\}REDO\{x_{new}\}$$

where DO' is a (possibly partial) execution of DO and x'_{new} is an intermediate value of the object due to partial execution.

- For physical logging, the UNDO and REDO trivially meet the above requirements.
- For logical (as well as physiological logging), the UNDO and the REDO may NOT satisfy the above requirements– hence violation of operation consistency poses a problem!

Problems With Logical Logging

- If logical logging is used, operation consistency may be violated.
- This introduces two problems in recovery: Consider a logical operation o consisting of suboperations o_1, o_2, o_3 .
 - **Partial Failure:** A failure occurs during the execution of o such that o_1 and o_2 have been reflected in the database but o_3 has not. How to UNDO the effects of partial operation execution.
 - *Action Consistency:* A system failure occurs such that the persistent copy of the DBMS contains effects of some subset of the operations o_1, o_2, o_3 but not of others. For example, effects of o_2 , and o_3 appear but not of o_1 . In this case, if the transaction commits, how to REDO the effects of o , or if the transaction did not commit, how to UNDO the effects of o_2 and o_3 from the persistent data copy.

Overcoming Operation Consistency Problems

- To address the above partial action and action consistency problems, logging and recovery done at 2 levels.
- Shadowing Strategy: Do not do update in place. Instead create shadow copies of the pages that are updated. If transaction aborts during the middle of the operation execution, the operation is not undone using the logical log record, but instead the page state is restored to the shadow copy. If action fully represented on the page, then undo using the logical log record. This is the scheme used in System R.
- Multi-level Strategy: Use some other mechanism (e.g., physiological/ physical logging) to implement atomicity of the action. Those physical/physiological logs used to undo partially executed actions. If action execution is complete, then use logical log record. This is referred to as multi-level recovery.
- Multi-level strategy results in larger number of log records than would be generated by Physical/physiological logging.
- Shadow paging has a large number of problems as well.
- However, since potentially higher concurrency results logical logging and logical UNDO are used in modern

recovery algorithms (see ARIES).

Physiological Logging

- Complex actions structured as a sequence of page level actions.
- Action consistency problem is also present if physiological logging used.
- Guaranteeing page action consistency: applications do recovery from partial failures during page update. another way is page invalidation. Invalidated page can always be recovered from the copy on persistent storage + log records
- A partially modified page is prevented from being flushed to the disk.
- This ensures that the persistent data is action consistent and hence REDO and UNDO using log records is possible on persistent data.
- Page during modification is locked using a semaphore to prevent others from seeing a partial update.

Issues in Implementing Recovery Based on Physiological Logging

- **FIX**: cover all page read and write using semaphore. Prevent partially modified pages from reaching the disk.
- **WAL**: force the page log record before overwriting persistent data copy.
- **FL@C**: force transaction log records as part of commit.

Implementing WAL and Force log at Commit

- WAL: Each page stores a pageLSN, where pageLSN is the log sequence number of the most recent update to the page. Before flushing page to disk, logs are flushed upto pageLSN to disk.
- FL@C: Logs upto the commit record are forced to disk, before the transaction commits.

Another Usage of PageLSN in Recovery based on Physiological Logging

- Recall that REDO and UNDO operations for physiological logging may not be idempotent.
- Consider restart after a system failure.
- During restart, we wish to REDO updates of committed transaction and UNDO changes of uncommitted transaction.
- Since REDO and UNDO are not idempotent, blindly REDOing updates will cause inconsistency (what if the change already present on persistent database). Similarly, blindly UNDOing will cause problems (what if change not present in persistent database).
- Whether or not an update appears on the page can be made testable by using a pageLSN.
- Since pageLSN is the value of the last log record to update the page, if the lsn of the log less than pageLSN, then the update appears on the page. Else, if lsn of the log is greater than pageLSN, then effect of the action does not appear on the page.

Implementing Roll_Back

- Roll_Back operation results in the system reading the log records of the transaction in reverse chronological order and undoing each operation.
- Recall each page maintains a state variable — pageLSN which reflects upto which point in the log the changes have appeared on the page. What should we do to the pageLSN value during rollback?
- The system, in order to undo the operation, writes a log record— compensation log record (CLR).
- The pageLSN of the page on which the update took place is modified to be the lsn of the (CLR).

Checkpointing

Periodically, the system execute a checkpoint.

Checkpoints are used to reduce the amount of REDO work done during restart.

low water mark is the location in the log from where the system begins its redo processing in presence of failures.

Checkpoints increase the low water mark thereby reducing the REDO work at restart.

- Sharp Checkpoints: System periodically checkpoint the state of volatile memory as a synchronous operation.
- Fuzzy Checkpoints: System checkpoint volatile memory as an asynchronous operations during normal processing. Fuzzy checkpoint write *begin_checkpoint* and *end_checkpoint* log records.

Checkpointing

An example of a sharp checkpoint is if the system writes out to stable storage the entire buffer cache as a synchronous operation. Such a checkpoint brings low water mark to the current LSN.

An example fuzzy checkpoint is if the RM writes out to stable storage the entire buffer cache as an asynchronous operation. Low water mark of resource manager is the LSN of the *begin_checkpoint* log record.

There are numerous optimizations:

Example: write out only modified pages, instead of writing pages to database write them to log,
two-checkpoint approach, indirect checkpointing.

Restart Implementation

Restart consists of three phases:

- Analysis Phase: starts from the transaction manager's last checkpoint and reads until end of log. Constructs a list of all active and committed transactions.
- Redo Phase: Starts from the Redo low water mark, redoing the effects of committed as well as transactions that were live at failure. (repeating history).
- Undo Phase: Undo the effects of the transactions that were live at the time of failure. Undo scan may result in CLRs to be written.

After the undo phase a new checkpoint is taken and the transaction processing begins.

Justification of Correctness of Restart

see Table 11.10 of GR.

Also, notice how restart ensures correctness of transactions that were in the middle of roll back at the time of failure.

For such transactions, the REDO of original records as well as the CLR's is performed.

Furthermore, during the UNDO phase, all the log records (both CLR's as well as the log records written during forward processing are undone).

Notice that this strategy may result in unbounded logging in the presence of multiple failures.

One way of preventing this is by chaining the CLR's to the forward log records. That is, a CLR for a log record LR is chained to the previous log record for the same transaction.

All the operations between the pointer from a CLR to the forward log record are not undone.

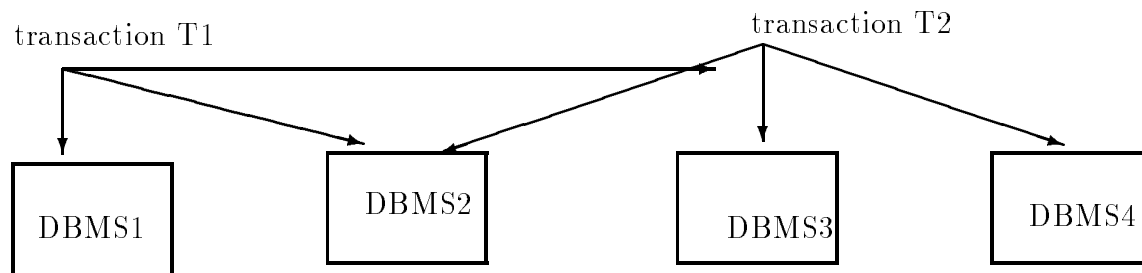
You should develop a complete restart strategy which implements this optimization.

Restart Issues

- Why is it important to repeat history? That is, do we need to Redo effects of transactions that did not commit?
 - Since the state of the page is identified by a single pageLSN, how will we represent the fact that a given log record's effect are not present on the page.
 - If multilevel recovery is used, then we may wish to undo at a logical level. Correctness of this will require that the redo at lower level be complete even if transaction aborts.
- Can the Undo pass be executed before the Redo pass?
 - Again, if logical undo is used during Undo pass, its correctness will require that the effects of the Redo be done before Undo.
 - Since a single state variable pageLSN used to identify the state of the page, and Undo results in writing CLRs, it will be difficult to trace the state of the page if Undo is done before Redo.

Restart Optimizations

- Accelerating Restart:
 - Frequent checkpoints.
 - perform restart incrementally– recover most critical resource first.
 - Exploit parallelism.
- Dealing with long transactions.
 - Abort such transaction.
 - Copy forward – copy the log records of such transactions forward.
 - Copy aside– copy the log records of such transactions to a side file.



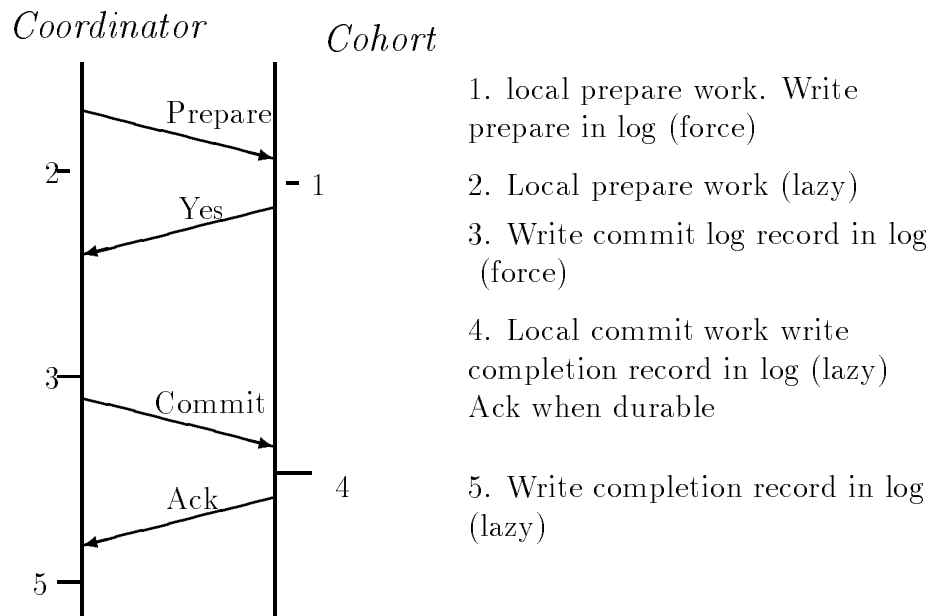
Distributed Transaction Processing

- A distributed DBMS consists of many local DBMSs connected by some communication system.
- Transactions may execute at multiple local DBMSs.
- Each local DBMS maintains a log and has recovery procedures for transaction rollback as well as restart.
- Consider a transaction that executes at multiple local DBMSs– say DBMS1, DBMS2, and DBMS3.
- If the transaction requests a rollback during execution, then the effects of the transaction can be undone by requesting rollback at all the DBMSs at which the transaction executed.
- What to do if a transaction requests commit work?
- If commit processing at sites is done asynchronously, multiple problems can arise.
 - If DBMS2 decides that committing the transaction violates its consistency and it decides to abort the transaction, loss of atomicity will occur in case

DBMS1 and DBMS3 have already committed the transaction.

- If DBMS2 fails and the log records for the transaction were not in persistent storage before failure, DBMS2 may not be able to commit the transaction on recovery resulting in violation of atomicity in case DBMS1 and DBMS3 committed the transaction.
- To guarantee atomicity, all DBMSs on which a transaction executes must use some agreement protocol to ensure that either the transaction commits at all the DBMSs or it aborts at all the DBMSs on which it executes.
- The protocol used must be resilient to site failures as well as communication failures (e.g., due to network partition, timeout failures).
- That is, the protocol must guarantee that in spite of all failures, if all failures are repaired, then all sites agree to either commit or abort the transaction.
- The agreement protocol that is most commonly used is some version of the *two-phase commit* protocol.

Two-Phase Commit Protocol



- The coordinator maintains a *protocol database* in main memory in which it maintains the progress of the transaction, info about cohorts, have cohorts responded yet, etc.

Two-Phase Commit Protocol– timeout actions

- At multiple stages during the execution of the 2PC, sites wait for messages from each other.
- If a message is not received within a bounded amount of time, the site executes a timeout and undertakes a timeout action.
- If a coordinator times out waiting for a vote from a cohort, it executes an ABORT, sends ABORT message to each participating cohort, deletes transaction from protocol database.
- If a cohort times out waiting for a prepare message from the coordinator, it executes an ABORT, and it sends an ABORT message to each site (it could have waited and replied NO when coordinator asked for vote).
- If the cohort times out waiting for decision from the coordinator, it cannot unilaterally abort the transaction. It instead forwards the transaction to the recovery process.
- The recovery process, periodically, executes a *status_transaction* call to the coordinator. Such a transaction is *blocked* for the recovery of the failure. Notice that the recovery process could have used

other termination protocols as well to reduce blocking (e.g., polling other cohorts).

- If the coordinator times out waiting for an ACK from a cohort, it forwards the transaction to the recovery process.
- The recovery process, periodically sends a *Commit* message to the cohort. Once the failure is repaired and all cohorts reply, the coordinator writes a completion record and the information from the protocol database is deleted.

Distributed Transaction Processing— Cohort and Coordinator Recovery

- A site recovers independently using the REDO and UNDO pass as discussed before.
- During the recovery, a site determines all transactions for which it has a prepared record, but no commit record. Notice, the site must be a cohort for such a transaction.
 - It reacquires all the locks for such a transaction and communicates with the coordinator of the transaction for the commitment of such a transaction.
 - The coordinator does the following: If it has no information about the transaction in its protocol database in main memory, then it sends back an ABORT message. If it has information and the transaction has committed, it sends back a commit. Else, if it has information but the transaction has not committed, it sends back a wait message.
- For each transaction for which it finds a commit record (and not a complete record), the site must be a coordinator for the transaction. It recreates an entry for the transaction in its protocol database and communicates with the cohorts for the completion of the transaction.

Two Phase Commit Optimizations

- Usage of 2PC results in blocking
 - If coordinator fails, after the cohort has send its YES vote, the cohort is stuck holding onto the locks and data. It cannot unilaterally abort the transaction.
 - If the communication between cohort and coordinator fails after the cohort has send its YES vote, the cohort is stuck holding onto the locks and data. It cannot unilaterally abort the transaction.
- If we ignore communication failures, then it is possible to design a protocol which is non-blocking inspite of the coordinator failure. (Skeen et. al. 83).
- Unfortunately, in presence of communication failures, it is impossible to design a protocol that is non-blocking.
- If generality of the transactions (that is, what data items they read and write) can be restricted, then it is possible to design a protocol which is non-blocking (Mehrotra et. al. 92).

Three Phase Commit Protocol

- Reason 2PC blocks – since operational site on timeout in prepare state does not know if the failed site(s) had committed or aborted the transaction. They could have done either. One possibility is to poll all the other operational sites if they know the decision. However, it is possible all operational sites are in doubt.
- Approach to Non-Blocking – make sure that:
 - There exists no local state such that its concurrency set contains both an abort and a commit state.
 - There exists no non-committable state whose concurrency set contains a commit state. (state is committable if occupancy of that state by any site implies that every site has voted yes to commit the transaction)
- Necessity of this condition illustrated by considering the case when only a single operational site is left. If either of the above is violated, blocking will result.
- Sufficiency of this condition can be illustrated by designing a termination protocol that under the assumption that the above holds can be used to terminate the protocol consistently at all operational sites.

Three Phase Commit

- 3PC termination protocol: Choose a backup coordinator from the remaining operational sites. Backup coordinator sends messages to other operational transaction cohorts to make a transition to its local state and waits for an acknowledgement. Based on its local state it makes a decision to commit or abort the transaction. It decides to commit, if the concurrency set of its local state contains a commit state. Else, it decides to abort the transaction.
- 3PC is non-blocking under the assumption that there are no communication failures and network partitions.
- In case of network partition, different partitions can take different decisions with respect to the commitment of a transaction. Hence 3PC cannot ensure atomicity.
- 3PC can be extended with a different termination protocol to ensure atomicity inspite of network partitions. However, it will also block.
- Alternative approach to non-blocking – Redo Approach!

Non-Blocking Two-Phase Commit

- The basic 2PC protocol can be made non-blocking by implementing the coordinator as a process-pair.
- In this scheme, the primary coordinator checkpoints its state to the backup coordinator.
- As long as one of the coordinators do not fail, the transaction does not block.
- Non-blocking 2PC is often used in practice for fault-tolerant systems.
- Most transaction processing systems use a *heuristic outcome* to overcome the problem of blocking!
- Atomicity is not guaranteed in all cases, and human intervention used to make the database consistent.

Two Phase Commit Optimizations

- 2PC imposes high overhead on normal transaction processing – number of messages, number of rounds, amount of disk writes.
- Various optimization strategies have been developed to reduce these overheads.
- Presumed abort protocol, presumed commit protocol, read only optimization, linear 2PC, transfer of commit, distributed 2PC, optimized presumed commit protocols.
- Current industrial implementations use presumed nothing and presumed abort protocols.

Transaction Management Summary

- So far in the course we have studied the basic transaction model– ACID properties.
- We have studied how ACID properties of computations are implemented both in a centralized as well as a distributed environment.
- Over the last decade, transaction processing technology has advanced very rapidly.
 - Replicated Data management in distributed Databases.
 - Remote backup systems for disaster recovery.
 - Transaction processing in Parallel Database Environments.
 - Transaction processing in active database systems.
 - Maintaining multiversion Data to support high read throughput.
 - Transaction processing in Client-server database environments.
 - Transaction processing in heterogeneous distributed database environments.
- Emerging applications however are pushing the existing transaction technology to its limits.
 - mobile environments.

- workflow computations.
- multimedia databases.
- Highly distributed Web based Applications.
- Cooperative work environments.
- Agent-based computing environments.