Unit 5

Recovery Concepts

Failure Classification

Failure has been categoried, as follows −

Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can’t go any further. This is called transaction failure where only a few transactions or processes are hurt.

Reasons for a transaction failure could be −

* **Logical errors** − Where a transaction cannot complete because it has some code error or any internal error condition.
* **System errors** − Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

System Crash

There are problems − external to the system − that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk Failure

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

* It should check the states of all the transactions, which were being executed.
* A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
* It should check whether the transaction can be completed now or it needs to be rolled back.
* No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

* Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
* Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows −

* The log file is kept on a stable storage media.
* When a transaction enters the system and starts execution, it writes a log about it.

<Tn, Start>

* When the transaction modifies an item X, it write logs as follows −

<Tn, X, V1, V2>

It reads Tn has changed the value of X, from V1 to V2.

* When the transaction finishes, it logs −

<Tn, commit>

The database can be modified using two approaches −

* **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
* **Immediate database modification** − Each log follows an actual database modification. That is, the database is modified immediately after every operation.

Recovery with Concurrent Transactions

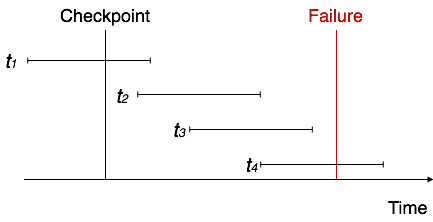
When more than one transaction are being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

Checkpoint

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

Recovery

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −



* The recovery system reads the logs backwards from the end to the last checkpoint.
* It maintains two lists, an undo-list and a redo-list.
* If the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>, it puts the transaction in the redo-list.
* If the recovery system sees a log with <Tn, Start> but no commit or abort log found, it puts the transaction in undo-list.

Shadow paging

* Shadow paging is a technique for providing atomicity and durability in database systems.

• Shadow paging is a copy-on-write technique for avoiding in-place updates of pages. Instead, when a page is to be modified, a shadow page is allocated.

• Since the shadow page has no references (from other pages on disk), it can be modified liberally, without concern for consistency constraints, etc. When the page is ready to become durable, all pages that referred to the original are updated to refer to the new replacement page instead. Because the page is "activated" only when it is ready, it is atomic.

• This increases performance significantly by avoiding many writes on hotspots high up in the referential hierarchy (e.g.: a file system superblock) at the cost of high commit latency.

Shadow paging considers:

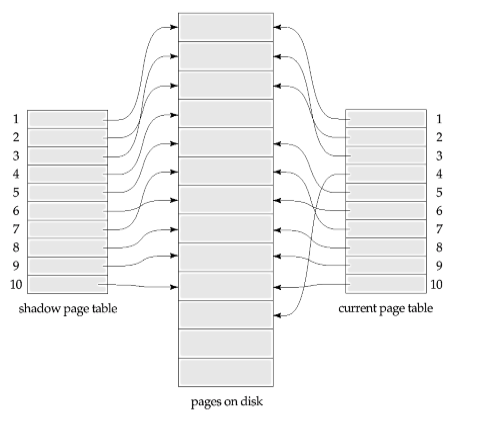
1. The database is partitioned into fixed-length blocks referred to as PAGES.
2. Page table has n entries – one for each database page.
3. Each contain pointer to a page on disk (1 to 1st page on database and so on…).

The idea is to maintain 2 pages tables during the life of transaction.

1. The current page table
2. The shadow page table

When transaction starts, both page tables are identical

1. The shadow page table is never changed over the duration of the transaction.
2. The current page table may be changed when a transaction performs a write operation.
3. All input and output operations use the current page table to locate database pages on disk.



**Advantages:**

• No Overhead for writing log records.

• No Undo / No Redo algorithm.

• Recovery is faster.

**Disadvantages:**

• Data gets fragmented or scattered.

• After every transaction completion database pages containing old version of modified data need to be garbage collected.

• Hard to extend algorithm to allow transaction to run concurrently.

**Aries Algorithm**

The ARIES recovery procedure consists of three main steps: analysis, REDO, and UNDO. The **analysis step** identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The REDO **phase** actually reapplies updates from the log to the database. Generally, the REDO operation is applied only to committed transactions. However, this is not the case in ARIES. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. Additionally, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and therefore does not need to be reapplied. Thus, *only the necessary REDO operations* are applied during recovery. Finally, during the UNDO **phase**, the log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. Additionally, checkpointing is used. These tables are maintained by the trans-action manager and written to the log during checkpointing.

In ARIES, every log record has an associated **log sequence number (LSN)** that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a *specific change* (action) of some transaction. Also, each data page will store the LSN of the *latest log record corresponding to a change for that page.*

A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end). The need for including the first three actions in the log has been discussed, but the last two need some explanation. When an update is undone, a *compensation log record* is written in the log. When a transaction ends, whether by committing or aborting, an *end log record* is written.

Common fields in all log records include the previous LSN for that transaction, the transaction ID, and the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write) action, additional fields in the log record include the page ID for the page that contains the item, the length of the updated item, its offset from the beginning of the page, the before image of the item, and its after image.

Besides the log, two tables are needed for efficient recovery: the **Transaction Table** and the **Dirty Page Table**, which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for *each active transaction,* with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The Dirty Page Table contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

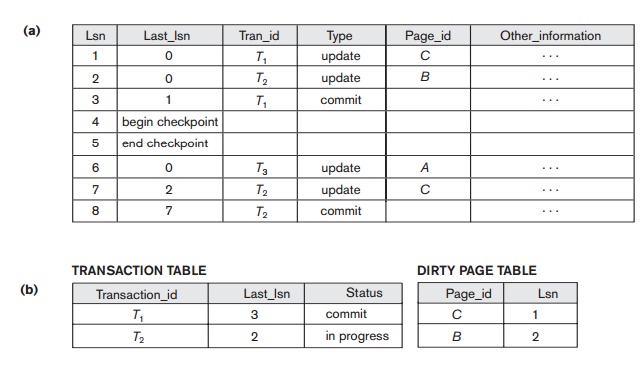
**Checkpointing**in ARIES consists of the following: writing abegin\_checkpointrecordto the log, writing an end\_checkpoint record to the log, and writing *the* *LSN* *of* the begin\_checkpoint record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the end\_checkpoint record, the con-tents of both the Transaction Table and Dirty Page Table are appended to the end of the log. To reduce the cost, **fuzzy checkpointing** is used so that the DBMS can continue to execute transactions during checkpointing . Additionally, the contents of the DBMS cache do not have to be flushed to disk during checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery. Note that if a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which is used for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The **analysis phase** starts at the begin\_checkpoint record and proceeds to the end of the log. When the end\_checkpoint record is encountered, the Transaction Table and Dirty Page Table are accessed (recall that these tables were written in the log during checkpointing). During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction *T* in the Transaction Table, then the entry for *T* is deleted from that table. If some other type of log record is encountered for a transaction *T* , then an entry for *T* is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page *P*, then an entry would be made for page *P* (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The REDO **phase** follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk.* It can determine this by finding the smallest LSN, *M,* of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to an LSN < *M,* for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN = *M* and scans forward to the end of the log. For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page *P* that is not in the Dirty Page Table, then this change is already on disk and does not need to be reapplied. Or, if a change recorded in the log (with LSN = *N*, say) pertains to page *P* and the Dirty Page Table contains an entry for *P* with LSN greater than *N*, then the change is already present. If neither of these two conditions hold, page *P* is read from disk and the LSN stored on that page, LSN(*P*), is compared with *N*. If *N* < LSN(*P*), then the change has been applied and the page does not need to be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the undo\_set—has been identified in the Transaction Table during the analysis phase. Now, the UNDO **phase**proceeds by scanning backward from the end of the log and undoing theappropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of trans-actions in the undo\_set has been undone. When this is completed, the recovery process is finished and normal processing can begin again.

Consider the recovery example shown in Figure 23.5. There are three transactions: *T*1,*T*2, and*T*3.*T*1updates page*C*,*T*2updates pages*B*and*C*, and*T*3updates page A.



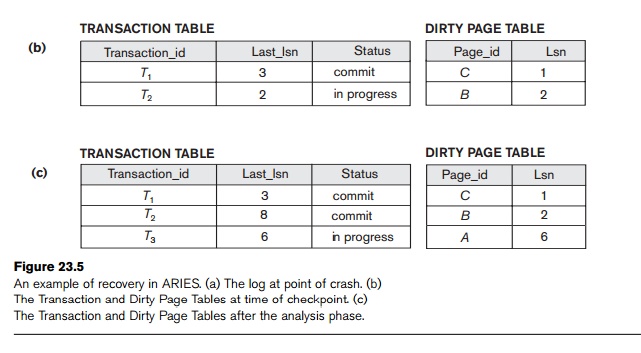


Figure 23.5(a) shows the partial contents of the log, and Figure 23.5(b) shows the contents of the Transaction Table and Dirty Page Table. Now, suppose that a crash occurs at this point. Since a checkpoint has occurred, the address of the associated begin\_checkpoint record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end. The end\_checkpoint record would contain the Transaction Table and Dirty Page Table in Figure 23.5(b), and the analysis phase will further reconstruct these tables. When the analysis phase encounters log record 6, a new entry for transaction *T*3 is made in the Transaction Table and a new entry for page A is made in the Dirty Page Table. After log record 8 is analyzed, the status of transaction *T*2 is changed to committed in the Transaction Table. Figure 23.5(c) shows the two tables after the analysis phase.

For the REDO phase, the smallest LSN in the Dirty Page Table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. The LSNs {1, 2, 6, 7} corresponding to the updates for pages C, B, A, and C, respectively, are not less than the LSNs of those pages (as shown in the Dirty Page Table). So those data pages will be read again and the updates reapplied from the log (assuming the actual LSNs stored on those data pages are less then the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure 23.5(c)), UNDO is applied only to the active transaction *T*3. The UNDO phase starts at log entry 6 (the last update for *T*3) and proceeds backward in the log. The backward chain of updates for transaction *T*3 (only log record 6 in this example) is followed and undone.

**Aggregate Data Models:**

An aggregate is a collection of data that we interact with as a unit. These units of data or aggregates form the boundaries for ACID operations with the database, Key-value, Document, and Column-family databases can all be seen as forms of aggregate-oriented database.

Aggregates make it easier for the database to manage data storage over clusters, since the unit of data now could reside on any machine and when retrieved from the database gets all the related data along with it. Aggregate-oriented databases work best when most data interaction is done with the same aggregate, for example when there is need to get an order and all its details, it better to store order as an aggregate object but dealing with these aggregates to get item details on all the orders is not elegant.



A *key-value database* (also known as a key-value store and key-value store database) is a type of [NoSQL](https://database.guide/what-is-nosql/) database that uses a simple key/value method to store data.

The key-value part refers to the fact that the database stores data as a collection of key/value pairs. This is a simple method of storing data, and it is known to scale well.

The key-value pair is a well established concept in many programming languages. Programming languages typically refer to a key-value as an *associative array* or *data structure*. A key-value is also commonly referred to as a *dictionary* or *hash*.

**Examples of Key-Value Stores**

Below are examples of key-value stores.

These are simple examples, but the aim is to provide an idea of the how a key-value database works.

**Phone Directory**

|  |  |
| --- | --- |
| **Key** | **Value** |
| Bob | (123) 456-7890 |
| Jane | (234) 567-8901 |
| Tara | (345) 678-9012 |
| Tiara | (456) 789-0123 |

**Document database**

A document database is a type of nonrelational database that is designed to store and query data as JSON-like documents. Document databases make it easier for developers to store and query data in a database by using the same document-model format they use in their application code. The flexible, semistructured, and hierarchical nature of documents and document databases allows them to evolve with applications’ needs. The document model works well with use cases such as catalogs, user profiles, and content management systems where each document is unique and evolves over time. Document databases enable flexible indexing, powerful ad hoc queries, and analytics over collections of documents.

In the following example, a JSON-like document describes a book.

[

{

"year" : 2013,

"title" : "Turn It Down, Or Else!",

"info" : {

"directors" : [ "Alice Smith", "Bob Jones"],

"release\_date" : "2013-01-18T00:00:00Z",

"rating" : 6.2,

"genres" : ["Comedy", "Drama"],

"image\_url" : "http://ia.media-imdb.com/images/N/O9ERWAU7FS797AJ7LU8HN09AMUP908RLlo5JF90EWR7LJKQ7@@.\_V1\_SX400\_.jpg",

"plot" : "A rock band plays their music at high volumes, annoying the neighbors.",

"actors" : ["David Matthewman", "Jonathan G. Neff"]

}

},

{

"year": 2015,

"title": "The Big New Movie",

"info": {

"plot": "Nothing happens at all.",

"rating": 0

}

}

]

**Column-Family Stores**

It is a tabular structure which it realized with sparse columns and no scheme Ex: Hbase, Cassender



Column-family databases organize their columns into column families. Each column has to be part of a single column family, and the column acts as unit for access, with the assumption that data for a particular column family will be usually accessed together.

This also gives you a couple of ways to think about how the data is structured.

• Row-oriented: Each row is an aggregate (for example, customer with the ID of 1234) with column families representing useful chunks of data (profile, order history) within that aggregate.

• Column-oriented: Each column family defines a record type (e.g., customer profiles) with rows for each of the records. You then think of a row as the join of records in all column families.

**Graph DataBases**

A *graph database management system* (henceforth, a *graph database*) is an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

There are two properties of graph databases we should consider when investigating graph database technologies:

The underlying storage

Some graph databases use *native graph storage* that is optimized and designed for storing and managing graphs. Not all graph database technologies use native graph storage, however. Some serialize the graph data into a relational database, an object-oriented database, or some other general-purpose data store.

The processing engine

Some definitions require that a graph database use *index-free adjacency*, meaning that connected nodes physically “point” to each other in the database.[2](https://www.oreilly.com/library/view/graph-databases-2nd/9781491930885/ch01.html#idm4202848) Here we take a slightly broader view: any database that from the user’s perspective *behaves* like a graph database (i.e., exposes a graph data model through CRUD operations) qualifies as a graph database. We do acknowledge, however, the significant performance advantages of index-free adjacency, and therefore use the term *native graph processing* to describe graph databases that leverage index-free adjacency.



In Figure 3.1 we have a web of information whose nodes are very small (nothing more than a name) but there is a rich structure of interconnections between them. With this structure, we can ask questions such as “find the books in the Databases category that are written by someone whom a friend of mine likes.”

Graph databases specialize in capturing this sort of information—but on a much larger scale than a readable diagram could capture. This is ideal for capturing any data consisting of complex relationships such as social networks, product preferences, or eligibility rules.

The fundamental data model of a graph database is very simple: nodes connected by edges (also called arcs). Beyond this essential characteristic there is a lot of variation in data models—in particular, what mechanisms you have to store data in your nodes and edges. A quick sample of some current capabilities illustrates this variety of possibilities: FlockDB is simply nodes and edges with no mechanism for additional attributes; Neo4J allows you to attach Java objects as properties to nodes and edges in a schemaless fashion ; Infinite Graph stores your Java objects, which are subclasses of its built-in types, as nodes and edges.

Once you have built up a graph of nodes and edges, a graph database allows you to query that network with query operations designed with this kind of graph in mind. This is where the important differences between graph and relational databases come in. Although relational databases can implement relationships using foreign keys, the joins required to navigate around can get quite expensive—which means performance is often poor for highly connected data models. Graph databases make traversal along the relationships very cheap. A large part of this is because graph databases shift most of the work of navigating relationships from query time to insert time. This naturally pays off for situations where querying performance is more important than insert speed.

**Schema less databases**

Properties

1. A schemaless database does not require conforming to a rigid schema (database, schema, data types, tables etc.) that one is required to live up to through the life of a system.
2. Does not enforce data type limitations on individual values pertaining to one single column type
3. Models the business usage and not a database schema, application or product.
4. Can store structured and unstructured data.
5. Eliminates the need to introduce additional layers (ORM layer) to abstract the relational model and expose it in an object oriented format.

Examble: MangoDb, Hbase

**Distributed Models**

The primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on. A more appealing option is to scale out—run the database on a cluster of servers. Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution. Depending on your distribution model, you can get a data store that will give you the ability to handle larger quantities of data, the ability to process a greater read or write traffic, or more availability in the face of network slowdowns or breakages.

Broadly, there are two paths to data distribution: replication and sharding.

Replication:

Data **Replication** is the process of storing data in more than one site or node. It is useful in improving the availability of data. It is simply copying data from a database from one server to another server so that all the users can share the same data without any inconsistency.

Sharding:

It is a computational storage technique in which large independent datasets are broken up into smaller units that are easier to manage. Doing so ensures a certain degree of data redundancy as well as an increase in overall performance. For example, handling all of the data required of a search engine within a single database may not be the most practical solution and if this data only exists on a single storage device, it can be easily lost or corrupted should the system encounter some degree of failure.

In the event a single node fails, replicated chunks of data can be retrieved and consolidated — mitigating risk because redundant data exists across a cluster of separate nodes. While one of the design motivations within distributed systems is to reduce single points of failure, [“almost all modern databases are natively sharded. Cassandra, HBase, HDFS, and MongoDB are popular distributed databases.”](https://medium.com/@jeeyoungk/how-sharding-works-b4dec46b3f6)