Investigate Database Recovery

Question 1 Answer

One scenario database recovery problem is when a system crashes due to hardware failure or power outage, resulting in the loss of committed transactions. In this scenario, it is essential to recover the database to a consistent state and ensure that all committed transactions are correctly reflected in the database.

To solve this problem, the ARIES recovery technique can be used. ARIES is an algorithm that provides fast and efficient recovery from a wide range of failures, including system crashes and media failures. The technique is based on the Write-Ahead Logging (WAL) protocol, which involves writing all changes made to the database to a log file before they are applied to the database.

To use ARIES, the administrator would first analyze the log file to determine which transactions were committed and which ones were not. They would then perform a redo pass to apply all committed transactions to the database, followed by an undo pass to roll back any uncommitted transactions that were in progress at the time of the failure.

ARIES is appropriate to use in environments where the database is *continuously updated* and *transactional consistency* is critical. It is optimized for high-speed, high-volume transaction processing systems and provides fast and efficient recovery from failures.

In conclusion, ARIES is an effective recovery technique for restoring lost data and ensuring transactional consistency in critical situations. It provides a robust and efficient way to recover data and maintain database integrity, making it an ideal solution for high-speed, high-volume transaction processing systems.

```
 let's assume that the system crashed during the middle of a transaction, resulting in a loss of committed transa
ctions. Here is an example of what the log table, transaction table, and dirty table might look like in this scen
ario:
```

Log Table: LSN Transaction ID Type Page ID Previous LSN Transaction Data 1 T1 Begin
2 T1 Update P1 1 P1: {a=1,b=2} 3 T1 Update P1 2 P1: {a=3,b=4} 4 T2 Begin
5 T2 Update P2 P2: {c=5,d=6} 6 T2 Commit 5
7 T1 Commit 3

Transaction Table: Transaction ID Status T1 Committed T2 Committed Dirty Table: Page ID Transaction ID RecLSN P1 T1 3 P2 T2 5

Assuming that the system crashed after LSN 5, but before the commit of transaction T2, we can see that transaction T2 was committed, but transaction T1 was not. To recover from this scenario, we would need to perform a redo pass to apply the committed changes made by T2 to the database. Then we would perform an undo pass to roll back the uncommitted changes made by T1. Consequently, after recovery, the Log Table, Transaction Table, and Dirty Table would look like this:

Log Table: LSN Transaction ID Type Page ID Previous LSN Transaction Data 1 T1 Begin
2 T1 Update P1 1 P1: {a=1,b=2} 3 T1 Update P1 2 P1: {a=3,b=4} 4 T2 Begin
5 T2 Update P2 P2: {c=5,d=6} 6 T2 Commit 5
7 T1 Abort 3

Transaction Table: Transaction ID Status T1 Aborted T2 Committed Dirty Table: Page ID Transaction ID RecLSN

As we can see, the changes made by T2 have been applied to the database, and the changes made by T1 have been rolled back. The Log Table, Transaction Table, and Dirty Table have been updated to reflect the recovered state of the database.

As we can see, the changes made by T2 have been applied to the database, and the changes made by T1 have been rolled back. The Log Table, Transaction Table, and Dirty Table have been updated to reflect the recovered state of the database.

```
library("RSQLite")
fpath="C:/Users/welcome/Downloads/"
dbfile="prosqi.db"
dbconn<- dbConnect(RSQLite::SQLite(),paste0(fpath,dbfile))
```

pragma foreign_keys = on enforces foreign keys. This is usually necessary because by default, SQLite does not enforce foreign keys.you could use R chunk or sql chunk to enable pragma.

```
dbExecute(dbconn, "PRAGMA foreign_keys = ON")
```

```
## [1] 0
```

Writing conditions to drop a table if the table already exists.

```
DROP TABLE IF EXISTS ModuleAuthorTracker
```

```
DROP TABLE IF EXISTS Module
```

```
DROP TABLE IF EXISTS Author
```

```
DROP TABLE IF EXISTS Affiliation
```

establishing a module-named table. Given that the term "number" is a reserved key word, it is not ideal for one of the attributes with the name "number" to have the same name. As a result, we changed number to "m number" and added other attributes as well.

```
CREATE TABLE Module(
m_number TEXT PRIMARY KEY NOT NULL,
title  TEXT NOT NULL,
lengthinMin INTEGER NOT NULL)
```

establishing a Author named table with the required attributes

```
CREATE TABLE Author(
aid INTEGER PRIMARY KEY NOT NULL,
name TEXT NOT NULL,
email TEXT NOT NULL,
affid INTEGER NOT NULL,
FOREIGN KEY(affid) REFERENCES Affiliation(affid)
 )
```

email_validation trigger is created as to check if the email is valid or not. here we are checking with generic form of email id where it should contain @ and website it is and containing a dot or not if any of these are violated then we get the error Invalid Emailid"

establishing a Affiliation named lookup table with the required attributes

```
CREATE TABLE Affiliation(
affid INTEGER PRIMARY KEY NOT NULL,
affiliation TEXT NOT NULL
check(affiliation in ("PT","FT","Contract"))
 )
```

The module and author tables has zero-many and many-many relationships and to decompose that we had a junction table hence we created a junction table named ModuleAuthorTracker having m_number and aid as both primary keys and foriegn keys.

```
CREATE TABLE ModuleAuthorTracker
(
m_number TEXT,
aid INTEGER,
PRIMARY KEY(m_number,aid),
FOREIGN KEY(aid) REFERENCES Author(aid),
FOREIGN KEY(m_number) REFERENCES Module(m_number)
 )
```

Inserting values into Module table

```
INSERT INTO Module VALUES
("a","Harrypotter",200),
("b","Saregama",300),
("c","dramatic",400)
```

Retrieving Data from Module table

```
select * from Module
```

3 records

m_number	title	lengthinMin
a	Harrypotter	200
b	Saregama	300
c	dramatic	400

Inserting values into Affiliation table

```
INSERT INTO Affiliation VALUES
(11,"PT"),
(22,"FT"),
(33,"Contract")
```

retrieving data from Affiliation table

```
select * from Affiliation
```

3 records

affid	affiliation
11	PT
22	FT
33	Contract

Inserting values into Author table

```
INSERT INTO Author VALUES
(1,"jayantha","dara.j@gmail.com",11),
(2,"jayatha","akuluri@gmail.com",22),
(3,"jayan","nanduri.j@gmail.com",33),
(4,"jayani","jayani@gmail.com",33)
```

```
INSERT INTO Author VALUES
(6,"jaha","dara.jgmail@com",11)
```

Data retrieving from Author table

```
select * from Author
```

5 records

aid	name	email	affid
1	jayantha	dara.j@gmail.com	11
2	jayatha	akuluri@gmail.com	22
3	jayan	nanduri.j@gmail.com	33
4	jayani	jayani@gmail.com	33
6	jaha	dara.jgmail@com	11

Inserting values into ModuleAuthorTracker table

```
INSERT INTO ModuleAuthorTracker(aid,m_number) VALUES
(1,"a"),
(1,"b"),
(1,"c"),
(2,"b"),
(3,"c")
```

```
INSERT INTO ModuleAuthorTracker(m_number,aid) VALUES
("c",2)
```

Retrieving data from ModuleAuthorTracker table

```
select * from ModuleAuthorTracker
```

6 records

m_number	aid
a	1
b	1
c	1
b	2
c	3
c	2

Retriving Title,author and affiliation data from all the tables in database.

```
select Module.title,Author.name,Affiliation.affiliation from Module
inner join ModuleAuthorTracker on Module.m_number=ModuleAuthorTracker.m_number
inner join Author on Author.aid=ModuleAuthorTracker.aid
inner join affiliation on Affiliation.affid=Author.affid
```

6 records

title	name	affiliation
Harrypotter	jayantha	PT
Saregama	jayantha	PT
dramatic	jayantha	PT
Saregama	jayatha	FT
dramatic	jayan	Contract
dramatic	jayatha	FT

Creating a function to check if the transactions are updated or not.

```
doUpdate <- function (dbcon,title,m_number,email,aid)
{
  txnFailed = FALSE

  dbExecute(dbcon, "BEGIN TRANSACTION")

  sql <- 'UPDATE Module SET title = ? WHERE m_number= ? ;'
  ps <- dbSendStatement(dbcon, sql,
                        params = list(title, m_number))
  if (dbGetRowsAffected(ps) < 1)
    txnFailed = TRUE
  dbClearResult(ps)

  sql <- 'UPDATE Author SET email= ? WHERE aid= ?;'
  ps <- dbSendStatement(dbcon, sql,
                        params = list(email,aid ))
  if (dbGetRowsAffected(ps) < 1)
    txnFailed = TRUE
  dbClearResult(ps)

  if (txnFailed == TRUE)
    dbExecute(dbcon, "ROLLBACK TRANSACTION")
  else
    dbExecute(dbcon, "COMMIT TRANSACTION")

  # return status, TRUE if successful; FALSE if failed
  return (!txnFailed)
}
```

Performing multiple updates in multiple tables

In this we are updating title and email of the Module and the author and we are giving valid inputs and hence the transaction is commited and transfer is successful

```
title<-"Game of thrones"
m_number<-"a"
email<-"gowrish28gog@gmail.com"
aid<-"3"

status <- doUpdate(dbcon1, title, m_number,email,aid)

if (status == TRUE) {
  cat('Transfer successful')
} else {
  cat('Transfer failed rollbacked')
}
```

```
## Transfer successful
```

In this case we have given input where the m_number doesnt exists and so the transaction rolbacks

```
title<-"harry potter"
m_number<-"e"
email<-"harshal@gmail.com"
aid<-"4"

status <- doUpdate(dbcon1, title, m_number,email,aid)

if (status == TRUE) {
  cat('Transfer successful')
} else {
  cat('Transfer failed rollback')
}
```

```
## Transfer failed rollback
```

```
dbDisconnect(dbcon1)
```