Investigate Database Recovery

Xuebao Zhao

2023-03-26

1. Choose a database recovery problem or scenario (perhaps from work) and then propose a solution using the techniques described in Chapter 11 in the textbook. Briefly describe the technique, when it is appropriate to use and what recovery problem it solves.

Suppose that a company has a financial system that stores transactional data related to its business operations. The financial system database used by the company contains tables for storing information about customers, accounts, transactions, and other financial data. The database is critical to the company's operations, as it is used for tracking sales, managing accounts, and generating financial reports.

To ensure the security and integrity of the data, the database is configured with various security measures, such as user authentication, encryption, and access controls. In addition, the database is backed up regularly to ensure that data can be recovered in the event of a failure or disaster.

The database is accessed by multiple users and applications, including a web-based front-end for customer transactions, as well as various backend systems for processing orders, managing inventory, and generating reports. As a result, the database is constantly being modified, with new transactions being added and updated on a regular basis.

In this case, a database server crash occurs due to hardware failure, resulting in the loss of important financial data. The WAL recovery technique can be used to recover the lost data. The WAL technique involves creating a log file that contains all modifications made to the database. Before any changes are written to the database, they are first written to the log file. This ensures that the log file always contains a record of all changes made to the database.

To use the WAL technique for recovery, the database is first restored to the most recent backup that is available. However, this backup will not contain any changes that were made after the backup was taken. To recover these changes, the log file is used to replay the transactions that were not yet committed at the time of the failure.

For example, suppose that the company had several financial transactions in progress at the time of the database crash. These transactions included purchases, refunds, and transfers of funds between accounts. The changes made to the database during these transactions were not yet committed, and therefore were lost due to the crash.

When using the WAL technique for recovery, the log file is used to replay these transactions, so that the lost data can be recovered. The log file contains a record of all changes made during the transactions, so these changes can be re-applied to the restored database to bring it up to date.

In this way, the WAL recovery technique can help organizations recover from database failures and ensure that their data remains safe and accessible. The WAL technique is appropriate to use when data loss must be avoided at all costs, such as in financial systems, and can solve the problem of lost or corrupted data due to database failure.

2. Using any of the SQLite database we have previously worked with, write an update that requires related modification of multiple tables and conduct those updates within a transaction. Test the code so that you show that the transaction works and write one test where the transaction fails and rolls back. This lesson shows how to work with transactions in SQLite:

Use MediaDB.db which is used in Assignment "DO: Query a Database with SQL" here.

Connect to database

```
library(RSQLite)

fpath = ""
dbfile = "MediaDB.db"
dbcon <- dbConnect(RSQLite::SQLite(), paste0(fpath,dbfile))
```

Write an update.

Assume that the UnitPrice of one track is mismarked. And now we need to change it in both tracks table and incoice_items table.

Create a transactions log table journal where all price changes are recorded

```
CREATE TABLE IF NOT EXISTS journal (
  ChangeId INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  TrackId INTEGER NOT NULL,
  amount DECIMAL NOT NULL,
  change_date DATE NOT NULL,
  FOREIGN KEY (TrackId) REFERENCES tracks(TrackId)
)
```

```
doTransfer <- function (dbcon, tracksid, trackitemsid, amount)
{
  txnFailed = FALSE

  dbExecute(dbcon, "BEGIN TRANSACTION")

  # Change the price in tracks table
  sql <- "UPDATE tracks
             SET UnitPrice = UnitPrice + ?
           WHERE TrackId = ?"
  ps <- dbSendStatement(dbcon, sql,
                        params = list(amount, tracksid))
  if (dbGetRowsAffected(ps) < 1)
    txnFailed = TRUE
```

```
  dbClearResult(ps)

  # credit destination account
  sql <- "UPDATE invoice_items
            SET UnitPrice = UnitPrice + ?
          WHERE InvoiceLineId = ?"
  ps <- dbSendStatement(dbcon, sql,
                        params = list(amount, trackitemsid))
  if (dbGetRowsAffected(ps) < 1)
    txnFailed = TRUE
  dbClearResult(ps)

  # add accounting transaction to log
  sql <- "INSERT INTO journal
           (TrackId,amount,change_date)
         VALUES (?, ?, date('now'))"
  ps <- dbSendStatement(dbcon, sql,
                        params = list(tracksid, amount))
  if (dbGetRowsAffected(ps) < 1)
    txnFailed = TRUE
  dbClearResult(ps)

  # Check if the price is greater than 0
  sql <- "SELECT UnitPrice FROM tracks
           WHERE TrackId = ?"
  ps <- dbSendStatement(dbcon, sql, params = tracksid)
  result <- dbFetch(ps)
  for(each in result$UnitPrice) {
    if (each < 0)
      txnFailed = TRUE
  }
  dbClearResult(ps)

  sql <- "SELECT UnitPrice FROM invoice_items
           WHERE InvoiceLineId = ?"
  ps <- dbSendStatement(dbcon, sql, params = trackitemsid)
  result <- dbFetch(ps)
  for(each in result$UnitPrice) {
    if (each < 0)
      txnFailed = TRUE
  }
  dbClearResult(ps)

  # commit transaction if no failure, otherwise rollback
  if (txnFailed == TRUE)
    dbExecute(dbcon, "ROLLBACK TRANSACTION")
  else
    dbExecute(dbcon, "COMMIT TRANSACTION")

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

```
dbGetQuery(dbcon, "SELECT TrackId, Name, UnitPrice FROM tracks WHERE TrackId = 1")
```

Related data which will be changed

```
##   TrackId                                    Name UnitPrice
## 1       1 For Those About To Rock (We Salute You)      0.99
```

```
dbGetQuery(dbcon, "SELECT InvoiceLineId, TrackId, UnitPrice FROM invoice_items WHERE TrackId = 1")
```

```
##   InvoiceLineId TrackId UnitPrice
## 1           579       1      0.99
```

Case that the transaction works

```
tracksid <- 1
trackitemsid <- 579
amount <- 1

status <- doTransfer(dbcon, tracksid, trackitemsid, amount)

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

```
## Transfer successful
```

```
dbGetQuery(dbcon, "SELECT TrackId, Name, UnitPrice FROM tracks WHERE TrackId = 1")
```

```
##   TrackId                                    Name UnitPrice
## 1       1 For Those About To Rock (We Salute You)      1.99
```

```
dbGetQuery(dbcon, "SELECT InvoiceLineId, TrackId, UnitPrice FROM invoice_items WHERE TrackId = 1")
```

```
##   InvoiceLineId TrackId UnitPrice
## 1           579       1      1.99
```

```
dbGetQuery(dbcon, "SELECT * FROM journal")
```

```
##   ChangeId TrackId amount change_date
## 1        1       1      1  2023-03-27
## 2        2       1      1  2023-03-27
## 3        3       1      1  2023-03-27
## 4        4       1      1  2023-03-28
## 5        5       1     -4  2023-03-28
## 6        6       1      1  2023-03-28
```

Case that the transaction fails and rolls back(UnitPrice can not smaller than 0)

```
tracksid <- 1
trackitemsid <- 579
amount <- -4

status <- doTransfer(dbcon, tracksid, trackitemsid, amount)

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

```
## Transfer failed
```

Disconnect the database

```
dbDisconnect(dbcon)
```