Investigate Database Recovery

Jiaxing Tan

2023-03-26

Question 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.

Scenario:

> A medium-sized e-commerce company The company has an online store that relies on a relational database management system (RDBMS) to store all product information, customer data, and transaction history.

Problem:

> One day, the database server experiences a system crash that results in data corruption. The whole application was malfuntional (downtime) due to the database failure and some data may loss.

Solutions:

> IT team could resort to the ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) technique to recover the database.

ARIES is based on write-ahead logging (WAL) and uses three main components: log sequence numbers (LSNs), a recovery log, and dirty page tables.

1. Analysis phase: In this phase, ARIES scans the recovery log forward from the most recent *checkpoint* to identify the state of each transaction at the time of the crash. It also constructs a dirty page table (DPT) to keep track of pages that were modified but not yet written to disk.

- Identify active transactions: ARIES determines which transactions were active (i.e., not yet committed or aborted) at the time of the crash. These transactions will need to be undone later.
- Update the dirty page table: For each log record with an update, ARIES updates the DPT with the corresponding page and the LSN of the first update to that page after the checkpoint.

2. Redo phase: During the redo phase, ARIES replays the recovery log from the oldest LSN found in the DPT to the end of the log. This ensures that all changes made by committed transactions are reflected in the database.

- Redo log records: For each log record, ARIES checks whether the affected page is in the dirty page table and whether the log record's LSN is greater than or equal to the page's LSN in the DPT. If both conditions are met, the log record's action is redone. This ensures that any changes made by committed transactions are reflected in the database.
- Update the DPT: As ARIES replays log records, it removes pages from the dirty page table once their updates have been redone.

3. Undo phase: In the undo phase, ARIES rolls back the active transactions identified during the analysis phase to ensure consistency in the database. It does this by processing the log records in reverse order.

- Identify log records to undo: ARIES selects the log records of active transactions that need to be undone.
- Perform undo actions: For each selected log record, ARIES performs the inverse operation specified in the log record to undo the changes made by the active transaction. For example, if the log record indicates an insertion, ARIES would delete the corresponding record. If the log record indicates an update, ARIES would restore the previous value.
- Log compensation records: As ARIES undoes log records, it writes compensation log records (CLRs) to the recovery log. CLRs include the LSN of the log record being undone, which allows ARIES to track the progress of the undo process and resume it correctly in case of another crash during the recovery.
- Mark transactions as aborted: Once all the log records for an active transaction have been undone, ARIES marks the transaction as aborted in the recovery log.

Once these steps are completed, the database can be brought back online and resume normal operations.

The ARIES recovery technique is appropriate to use in situations where a database system needs to recover from failures while maintaining consistency, durability, and minimizing the amount of redundant work during the recovery process. Some specific situations when it is appropriate to use ARIES include: system crashes, transaction aborts, media failures.

It is important to note that ARIES is most suitable for databases that use write-ahead logging (WAL) and have support for log sequence numbers (LSNs), recovery logs, and dirty page tables.

Citation: ITL Education Solutions Limited. (2008). Introduction to Database Systems. Pearson India.

Question 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.

```
library(pacman)
p_load(RSQLite)
p_load(sqldf)
p_load(tidyverse)
```

```
dbfile = "MediaDB.db"

conn <- dbConnect(RSQLite::SQLite(),dbfile)
```

Check constrains

```
sqlStatement <- "
SELECT sql
FROM sqlite_schema
WHERE name = 'tracks';
"
bs <- dbGetQuery(conn, sqlStatement)
bs
```

```
##
sql
## 1 CREATE TABLE "tracks"(\r\n\t"\n    [TrackId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,\r\n    [Name] NVARCH
AR(200)  NOT NULL,\r\n    [AlbumId] INTEGER,\r\n    [MediaTypeId] INTEGER  NOT NULL,\r\n    [GenreId] INTEGER,\r
\n    [Composer] NVARCHAR(220),\r\n    [Milliseconds] INTEGER  NOT NULL,\r\n    [Bytes] INTEGER,\r\n    [UnitPric
e] NUMERIC(10,2)  NOT NULL,\r\n    FOREIGN KEY ([AlbumId]) REFERENCES "album" ([AlbumId]) \r\n\t\tON DELETE NO A
CTION ON UPDATE NO ACTION,\r\n    FOREIGN KEY ([GenreId]) REFERENCES "genres" ([GenreId]) \r\n\t\tON DELETE NO AC
TION ON UPDATE NO ACTION,\r\n    FOREIGN KEY ([MediaTypeId]) REFERENCES "media_types" ([MediaTypeId]) \r\n\t\tON
DELETE NO ACTION ON UPDATE NO ACTION\r\n)
```

One wanna add a new tracks could call following function

```
addNewTrack <- function (dbcon, name, artistName, albumName, genreName, mediaTypeName,
                         composer, milliseconds, bytes, unitPrice) {
  addFailed = FALSE
  artistID = 0
  albumID = 0
  genreID = 0
  mediaID = 0


  # Some value is not nullable
  if (is.null(name) || is.null(milliseconds) || is.null(unitPrice)) {
    print("Detected illegal null value.")
    return (FALSE)
  }

  # album cannot be null value if artist exists, vice versa. But they can both be null
  if (is.null(artistName) || is.null(albumName)) {
    if (!(is.null(artistName) && is.null(albumName))) {
      print("Album cannot be null value if artist exists, vice versa.")
      return (FALSE)
    }
  }

  dbExecute(dbcon, "BEGIN TRANSACTION")

  # If artistName is not null
  if (!is.null(artistName)) {
    # Check if artist exists
    sqlStatement <- "
      SELECT * FROM artists WHERE name = ?
    "
    ps <- dbGetQuery(dbcon, sqlStatement, params = list(artistName))
    # if not exists, insert a new one
    if (nrow(ps) < 1) {
      sqlStatement <- "
        INSERT INTO artists
        (Name)
        VALUES
        (?)
      "
      ps <- dbSendStatement(dbcon, sql, params = list(artistName))
      if (dbGetRowsAffected(ps) < 1) addFailed = TRUE
      dbClearResult(ps)

      sqlStatement <- "
        SELECT ArtistId FROM artists WHERE name = ?
      "
      ps <- dbGetQuery(dbcon, sqlStatement, params = list(artistName))
    }

    # Get artists id
    artistID <- ps$ArtistId[1]
  }

  # if albumName is not null
  if (!is.null(albumName)) {
    # Check if album exists
    sqlStatement <- "
      SELECT * FROM albums WHERE title = ?
    "
    ps <- dbGetQuery(dbcon, sqlStatement, params = list(albumName))
    # if not exists, insert a new one
    if (nrow(ps) < 1) {
      sqlStatement <- "
        INSERT INTO albums
        (Title, ArtistId)
        VALUES
        (?, ?)
      "
      ps <- dbSendStatement(dbcon, sql, params = list(albumName, artistID))
      if (dbGetRowsAffected(ps) < 1) addFailed = TRUE
      dbClearResult(ps)

      sqlStatement <- "
        SELECT albumId FROM albums WHERE title = ?
      "
      ps <- dbGetQuery(dbcon, sqlStatement, params = list(albumName))
    }

    # If the existed album record has a different artists id
    if (ps$ArtistId[1] != artistID) addFailed = TRUE

    # Get albums id
    albumID <- ps$AlbumId[1]
  }

  # If genreName is not null
  if (!is.null(genreName)) {
    # Check if genre exists
    sqlStatement <- "
      SELECT * FROM genres WHERE name = ?
    "
    ps <- dbGetQuery(dbcon, sqlStatement, params = list(genreName))
    # if not exists, insert a new one
    if (nrow(ps) < 1) {
      sqlStatement <-
        INSERT INTO genres
        (Name)
        VALUES
        (?)
      "
      ps <- dbSendStatement(dbcon, sql, params = list(genreName))
      if (dbGetRowsAffected(ps) < 1) addFailed = TRUE
      dbClearResult(ps)

      sqlStatement <- "
        SELECT genreId FROM genres WHERE name = ?
      "
      ps <- dbGetQuery(dbcon, sqlStatement, params = list(genreName))
    }

    # Get genres id
    genreID <- ps$GenreId[1]
  }

  # if mediaTypeName is not null
  if (!is.null(mediaTypeName)) {
    # Check if mediatype exists
    sqlStatement <- "
      SELECT * FROM media_types WHERE name = ?
    "
    ps <- dbGetQuery(dbcon, sqlStatement, params = list(mediaTypeName))
    # if not exists, insert a new one
    if (nrow(ps) < 1) {
      sqlStatement <- "
        INSERT INTO media_types
        (Name)
        VALUES
        (?)
      "
      ps <- dbSendStatement(dbcon, sql, params = list(mediaTypeName))
      if (dbGetRowsAffected(ps) < 1) addFailed = TRUE
      dbClearResult(ps)

      sqlStatement <- "
        SELECT mediaTypeId FROM media_types WHERE name = ?
      "
      ps <- dbGetQuery(dbcon, sqlStatement, params = list(mediaTypeName))
    }

    # Get mediaTypes id
    mediaID <- ps$MediaTypeId[1]
  }


  # Ready for update tracks table
  sqlStatement <- "
    INSERT INTO tracks
    (Name, AlbumId, MediaTypeId, GenreId, Composer, Milliseconds, Bytes, UnitPrice)
    VALUES
    (?,?,?,?,?,?,?,?)
  "
  artistID <- if (artistID == 0) NULL else artistID
  albumID <- if (albumID == 0) NULL else albumID
  genreID <- if (genreID == 0) NULL else genreID
  mediaID <- if (mediaID == 0) NULL else mediaID

  ps <- dbExecute(dbcon, sqlStatement, params = list(name, albumID, mediaID,
                                                     genreID, composer,
                                                     milliseconds, bytes, unitPrice))

  if (ps < 1) addFailed = TRUE

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

  # return status: TRUE if successful; FALSE if failed
  return (!addFailed)

}
```

Test

1. Add a new track in a existed album

```
addNewTrack(conn, "Princess of the Dawn version 3", "Accept", "Restless and Wild", "Rock", "Protected AAC audio f
ile", "Deaffy & R.A. Smith-Diesel", 889998, 1212, 0.99)
```

```
## [1] TRUE
```

```
select * from tracks where name = 'Princess of the Dawn version 3'
```

1 records

TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice
3504	Princess of the Dawn version 3	3	2	1	Deaffy & R.A. Smith-Diesel	889998	1212	0.99

2. Add a new track that has conflict artist name and album name

```
addNewTrack(conn, "Princess of the Dawn version 4", "Aerosmith", "Restless and Wild", "Rock", "Protected AAC audi
o file", "Deaffy & R.A. Smith-Diesel", 889998, 1212, 0.99)
```

```
## [1] FALSE
```

```
select * from tracks where name = 'Princess of the Dawn version 4'
```

0 records

TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice

```
dbDisconnect(conn, )
```