Database Recovery Problem - Transaction Failure and Log Based Recovery

Mansi Pravin Thanki - 002128043

Problem Statement:

One of the most frequent type of failure is the transaction failure. This may lead to ACID properties not being exhibited by the database. Having incomplete transactions before failure and not recovering the correct state of system can lead to inconsistencies. Therefore, it is necessary to have a mechanism which can keep track of transactions and help to recover from the transaction failure.

Proposed Solution:

Log-based recovery: Similar to the old-school method where ledgers are maintained to keep the records of financial transactions, in system, we maintain logs of all the transactions. Each and every update operation on the database is tracked. Additionally, the logs are kept in a stable, non-volatile storage. The log based recovery techniques are mainly classified into following 2 types:

A. Deferred modification: The changes or transaction are all recorded in the logs, but it defers all the writes after the partial commit.

1. Let us make an assumption that the transactions will be executed sequentially in an order given to them 1,2,3…and so on
2. <T_i, start> is recorded to the log
3. For write operation: log <T_i, old value, new value> is generated
4. The write for the old value is deferred.
5. Whenever the transaction T_i commits partially, <T_i,commit> is written to log
6. If only <T_i,start> and <T_i, commit> are present in the log, the transaction has to be redone.
7. Consider an example of two transactions T_0 and T_1 where T_0 executes before T_1 If log contains <T_0,commit>, redo operation is needed If both transactions have such record: <T_0, commit> and <T_1, commit> then, redo will be performed followed by <T_1,commit>

At the time of transaction commit, it performs updates to buffer/disk.

Advantage of this scheme: Makes some aspects of the dataabase recovery simpler

Disadvantage: Overhead of storing local copy and can be a problem when the database is huge.

B. Immediate modification: It permits the update operation of a transaction that is uncommitted whenever the writes are issued. Due to presence of undo operation, the update log are required to have the old and new value. Before the database item is written, is is necessary that it is added to update log prior to it. There are 2 main operations involved:

a. undo(Ti): As the name suggests, it brings assigns the value of data updated by transaction Ti to their old values. Movement in reverse direction i.e backward from the last log record for Ti before the failure.

b. redo(Ti): new values by Ti are assigned to the previous values of data items. Movement in forward direction i.e forward from the initial log record for Ti before the failure. Always, the undo operation would be executed before the execution of redo operation. When recovering after failure:

1. if and no : then undo operation

2. if and : redo operation

Undo/Redo Algorithm:

The recovery manager performs the following during the recovery:

a. if transaction in active table: undo
b. if transaction in commit table

This redo step reperforms all the original steps that brings back the old values. This may seem as wastage of resource but the recovery process is simplified

When is deferred modification appropriate to use? In deferred modification, the changes or transaction are all recorded in the logs, but it defers all the writes after the partial commit.Therefore, if transaction failed before reaching the commit stage, no need to perform the undo.

Problems solved using deferred modification It helps maintain the ACID property of the database. The state is consistent, atomicity is maintained.

##### Q2 TRANSACTIONS ############

```
library(RSQLite)
file_path = (getwd())
dbfile = "/MediaDB.db"
dbcon <- dbConnect(RSQLite::SQLite(), paste0(file_path,dbfile))
```

INVOICES table

```
SELECT * FROM invoices
ORDER BY InvoiceId DESC
```

Table 1: Displaying records 1 - 10

InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total
412	58	2013-12-22 00:00:00	12,Community Centre	Delhi	NA	India	110017	1.99
411	44	2013-12-14 00:00:00	Porthaninkatu 9	Helsinki	NA	Finland	00530	13.86
410	35	2013-12-09 00:00:00	Rua dos Campeões Europeus de Viena, 4350	Porto	NA	Portugal	NA	8.91
409	29	2013-12-06 00:00:00	796 Dundas Street West	Toronto	ON	Canada	M6J 1V1	5.94
408	25	2013-12-05 00:00:00	319 N. Frances Street	Madison	WI	USA	53703	3.96
407	23	2013-12-04 00:00:00	69 Salem Street	Boston	MA	USA	2113	1.98
406	21	2013-12-04 00:00:00	801 W 4th Street	Reno	NV	USA	89503	1.98
405	20	2013-11-21 00:00:00	541 Del Medio Avenue	Mountain View	CA	USA	94040-111	0.99

InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total
404	6	2013-11-13 00:00:00	Rilská 3174/6	Prague	NA	Czech Republic	14300	25.86
403	56	2013-11-08 00:00:00	307 Macacha Güemes	Buenos Aires	NA	Argentina	1106	8.91

invoice_items table

```
SELECT * FROM invoice_items
ORDER BY InvoiceLineId DESC
```

Table 2: Displaying records 1 - 10

InvoiceLineId	InvoiceId	TrackId	UnitPrice	Quantity
2240	412	3177	1.99	1
2239	411	3163	0.99	1
2238	411	3154	0.99	1
2237	411	3145	0.99	1
2236	411	3136	0.99	1
2235	411	3127	0.99	1
2234	411	3118	0.99	1
2233	411	3109	0.99	1
2232	411	3100	0.99	1
2231	411	3091	0.99	1

Beginning transaction block

```
BEGIN TRANSACTION;
```

Inserting valid values in invoices

```
INSERT INTO invoices (InvoiceId, CustomerId, InvoiceDate,
BillingAddress,BillingCity,BillingState,
Billingcountry,BillingPostalCode, Total)
VALUES (413,1,date('now'),'Boston','Boston',
'MA', 'USA' ,'02120', 2.22); -- all valid values
```

Inserting valid values in invoice_items

```
INSERT INTO invoice_items (InvoiceLineId, InvoiceId,
TrackId,UnitPrice, Quantity)
VALUES (2241,413,3177,2.22,1); -- all valid values
```

Inserting valid values in invoice_items

```
commit transaction;
```

checking to see if above entry is seen in the invoice table

```
SELECT * FROM invoices WHERE InvoiceId= 413
```

Table 3: 1 records

InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total
413	1	2022-07-20	Boston	Boston	MA	USA	02120	2.22

checking to see if above entry is seen in the invoice_items table

```
SELECT * FROM invoice_items WHERE InvoiceId= 413
```

Table 4: 1 records

InvoiceLineId	InvoiceId	TrackId	UnitPrice	Quantity
2241	413	3177	2.22	1

Beginning transaction block

```
BEGIN TRANSACTION;
```

Inserting valid values in invoices

```
INSERT INTO invoices (InvoiceId, CustomerId, InvoiceDate,
BillingAddress,BillingCity,BillingState,
Billingcountry,BillingPostalCode, Total)
VALUES (414,1,date('now'),'Boston1','Boston1','MA1', 'USA1' ,'02120', 2.22);
```

Inserting invalid values in invoice_items -> NULL in trackid which conflicts the NOT NULL constraint

```
INSERT INTO invoice_items
(InvoiceLineId, InvoiceId, TrackId,UnitPrice, Quantity)
VALUES (2246,414,NULL,2.22,1);
```

```
## Error: NOT NULL constraint failed: invoice_items.TrackId
```

rolling back to previous state

```
ROLLBACK;
```

checking to see if above entry is not seen in the invoice table

expected behaviour: there should be no entry in the invoice table

```
SELECT * FROM invoices WHERE invoiceid=414
```

Table 5: 0 records

InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total

checking to see if above entry is not seen in the invoice table

expected behaviour: there should be no entry in the invoice_items table

```
SELECT * FROM invoice_items WHERE invoiceid=414
```

Table 6: 0 records

InvoiceLineId	InvoiceId	TrackId	UnitPrice	Quantity

```
dbDisconnect(dbcon)
```