# 하이퍼 큐브에서의 Write Ahead Log를 이용한 회복모듈의 설계및 구현

○ 최창락*, 송해상, 김탁곤, 박규호
* 한국과학기술원 전기및 전자공학과

# Design and Implementation of Recovery Module using Write Ahead Log

Choi, Chang-Rak*, Song, Hae-Sang, Kim, Tak-Gon, Park, Kyo-Ho
Dept. of Electrical Engineering, KAIST

## Abstract

Recovery module is one of the most important parts in database management system that makes database consistent. In this paper We describe problems and solutions that are issued at designing recovery module and explain how we implement recovery module on hypercube computer database system, COREDB.

## 1. 서론

현재 진행하고 있는 COREDB 프로젝트는 1990년부터 시작되어 하이퍼큐브 컴퓨터에서 병렬 관계형 데이타베이스 관리 시스템의 개발을 목표로 1993년까지 진행될 예정이다. 이 프로젝트의 목적은 가격 대비 성능면에서 우수하고 확장성이 뛰어난 데이타베이스 시스템의 설계와 구현이었다. 시스템은 각각 자신의 디스크를 가지는 노드 컴퓨터들이 하이퍼큐브 통신망으로 연결된 형태로 구성된다. 현재 그림 1.1.과 같은 큐브매니저와 8 노드 및 8 디스크를 갖는 3차원 하이퍼큐브를 개발중에 있다.

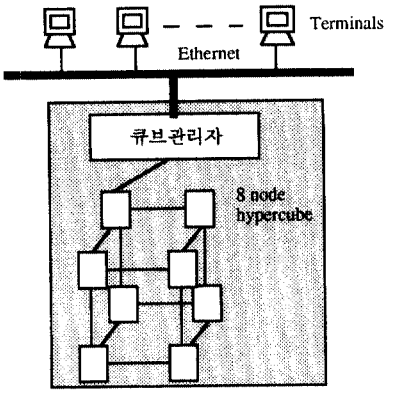


그림1.1. 8노드 하이퍼큐브의 구조

COREDB는 크게 3가지 파트로 나뉘어지는데, 해석파트, 실행파트, 저장구조 파트로 나뉘어진다. 이 중에서 저장구조 파트를 세부적으로 3개의 모듈로 나눌 수가 있다. 3개의 모듈은 동시성 제어 모듈, 회복 모듈, 저장 모듈로 나뉘어진다.

동시성 제어 모듈은 공유 데이타베이스의 액세스를 스케쥴한다. COREDB에서는 엄정 2단계 잠금 기법을 사용하였다. 잠금의 단위는 페이지 단위 잠금까지 하였지만, 레코드 단위 잠금으로의 확장을 고려하여 설계되었다.

회복 모듈은 시스템의 고장으로 인하여 데이타베이스의 일관성과 영속성이 깨지는 것을 방지하기 위하여 항상 복사본을 만들어 차후 재실행시에 이 복사본을 이용하여 고장전의 상태로 돌아갈 수 있게 한다.

이 회복 자료를 관리하는 방법은 여러가지가 있는데, 대표적인 방법으로는 shadow paging기법과 WAL(Write Ahead Log)방식이 있다.[1]

shadow paging기법은 작업을 하는 데이타 페이지와 그 복사본인 shadow 페이지를 가지고 있다. 이 방법은 취소나 재실행시에 빠르게 복구가 되는 반면에 다중 사용자 지원이나, 부분취소기능 등을 수행할 수가 없다. [3]

WAL기법은 데이타페이지를 변경, 삽입, 삭제할 때마다 이전의 데이타값과 연산수행 이후의 데이타값을 로그(Log)라고 불리는 화일에 저장을 시키고, 차후 고장발생시에는 로그를 참조하여서 복구를 한다. 이 기법은 로그정보를 추가함으로써, 추후 기능을 추가하기가 용이한 반면에, 로그를 기록하고 읽고 하는데 비용이 많이 든다.

## 2. 회복 모듈의 설계시 문제점과 해결방안

회복모듈을 만드는데 있어, 어떻게 프로그램을 짜고, 어떠한 기능을 넣느냐에 따라서 그 효율성이 많이 좌우된다. 여기서 설명하고 있는 문제점과 해결방안은 COREDB의 회복모듈 설계시에 고려되었다.

### 2.1. 간단성

대체로 동시성제어와 회복 관리자의 알고리즘은 복잡하게 되어 있다. 그렇기때문에 되도록이면 간단하게 구성하여, 구현상에서 실수등이 발생할 수 있는 요소를 줄여야 하고, 나중에 디버깅을 하기 편하게 한다.

### 2.2. 연산로깅

저장 시스템에서는 페이지 정리할때나, 인덱스 화일의 변경할때에 페이지의 내용이 아주 많이 바뀌게 되는데, 이 많은 부분이 변한것을 로그에 기록하게 되면 로그의 양이 많아지게 된다. 이런 이유때문에 연산 로깅이 필요하게 되는데, 이 기능은 연산에 대한 것을 로그에 기록하므로, 로그의 양을 줄일 수 있다.

### 2.3. 재실행의 병렬화

재실행하는 경우에 로그의 양이 많으면 그만큼 오랜 시간이 걸리게 된다. 시간이 적게 들게 하기위해서는 재실행을 수행하는 것

을 병렬화시키도록하여 재실행시 속도를 빠르게 한다.

### 2.4. 여러 단위 잠금자 지원

잠금 단위를 여러개를 지원하므로서 동시성제어의 효율을 높이게 한다. 잠금단위를 화일단위부터 레코드 단위까지 지원할 수 있도록 하여, 데이타의 종류에 따라서 잠금단위를 달리함으로써 시스템의 성능을 높일 수가 있다.

### 2.5. 점검점

점검점이 필요할때, 다른 프로세스의 활동에 영향을 주지 않고 실행할 수 있도록, 점검점 프로세스를 두고, 트랜잭션 프로세스의 활동과는 관계없이, 자신이 반영시켜야 하는 페이지들의 정보를 갖는다. 이것을 위해서 변경된 페이지들에 대한 정보를 갖는 테이블을 관리한다. 이 테이블을 참조해서 점검점 프로세스가 디스크에 반영할 페이지들에 래치를 걸어, 다른 트랜잭션 프로세스가 쓸 수 없도록 한 후, 디스크에 반영을 시킨다.

이렇게 하면 점검점이 일어난 후에 트랜잭션 일관성이나 연산 일관성은 보장되지 않게 된다. 이 일관성이 보장이 되지 않으면 재실행할때 모든 로그를 다 찾아야 하기때문에, 점검점을 실행한 의미가 별로 없어진다. 그래서 점검점이 끝났을때는 변경된 페이지 테이블과, 트랜잭션 테이블을 로그에 저장한다. 이 정보를 가지고 재실행시에 모든 로그를 다 읽지 않고, 필요한 내용만을 읽어서 회복을 시킬 수가 있게 된다.

### 2.6. 데이타 페이지의 디스크 반영

로그를 기록하는데, 매번 디스크에 반영시키는 오버헤드를 막기 위해서,로그 페이지를 버퍼에다 상주시킨다. 이런 경우 발생하는 문제가 있는데, 하나는 Commit시에 일어나는 일과, 다른 하나는 임의의 데이타 페이지가 디스크에 반영될 때이다.[5]

그림 2.1에서처럼 로그가 아직 디스크에 반영되지 않고, 해당 데이타 페이지가 디스크에 반영된후 시스템 고장이 발생하게 되면 그 해당 데이타 페이지의 취소 데이타는 로그 화일에 없게 되므로 회복할 수가 없게 된다. 이에 대한 해결책은 다음과 같이 두가지가 있게 된다. 첫번째 방법은 해당 페이지를 디스크에 반영하기 전에 로그 페이지를 디스크에 반영하는 방법이고 두번째 방법은 해당 페이지를 로그가 디스크에 반영될때까지 절대 디스크에 반영시키지 않게 하는 방법이다.

COREDB에서는 첫번째 방법을 쓰기로 하였다. 디스크에 반영되지 않은 로그 페이지의 첫번째 LSN과 비교하여 그 LSN보다 큰 데이타 페이지가 디스크에 반영될때는 먼저 로그를 디스크에 반영한 후에 데이타 페이지를 디스크에 반영하도록 한다.

### 2.7. 로그 페이지 래치

로그 페이지에 로그 레코드를 쓰기 위해서는 우선 래치를 걸고서 쓰게 된다. 이렇게 함에 따라서 로그 페이지를 엑세스를 할려는 트랜잭션 프로세스간에 경쟁이 일어나게 되고, 그 경쟁에서 지게 되는 프로세스는 기다리게 된다. 이런것때문에 상당히 오랜 지연이 생기게 된다.

우리가 구현한 데이타베이스는 기본 시스템이 유닉스를 쓰기때문에 이 문제에 대해서 많은 생각을 해야했다. 그런 오버헤드를 줄이면서 회복기법을 좀더 좋게 하는 방법을 연구했는데, 그중의 하나가 로그페이지를 볼륨마다 하나씩 두는 방법이다. 이렇게 함에 따라서 로그 데이타는 양이 많아지지만, 로그를 기록하는데 래치를 거는데 충돌이 줄고, 재실행할때는 볼륨마다 프로세스가 떠서 재실행을 함으로써 병렬처리할 수가 있기때문에 이득을 볼 수가 있다.

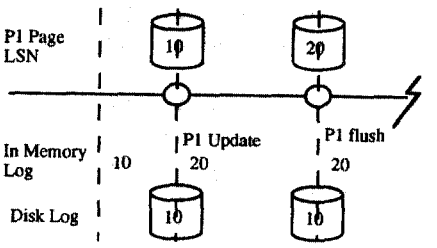


그림 2.1. In Memory에서 일관성 문제

### 2.8. 부분취소

CLR(Compensation Log Record)의 개념을 도입해서 사용하는데, 이 이유중 하나가 바로 부분취소를 고려해서이다. CLR은 부분취소를 하는 경우 취소를 한 로그 레코드를 대치하는 역할을 한다. 로그 화일은 계속 증가하면서 기록하는 것이므로 CLR을 이용해서 보정을 한다.

그림 2.2.에서와 같이 부분 취소를 하게 되면 앞으로 나가면서, CLR을 기록하게 되는데 이 CLR 레코드는 다른 로그 레코드와는 똑같지만, CLR은 다시 취소를 할 수가 없고, UndoNextLSN이 보통의 로그 레코드와는 달리 취소된 로그 레코드의 UndoNextLSN과 같게 된다.

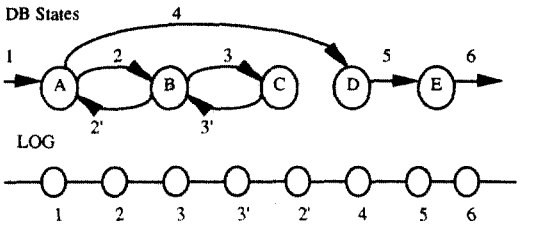


그림 2.2. 부분취소를 할때의 로그 기록

### 2.9. 로그 화일의 스페이스 관리

로그 스페이스하고 밀접한 관련이 있는것은 점검점인데, 이 점검점이 일어나는 시점의 선택을 로그의 스페이스하고 관련지어서 해야한다. 하지만 점검점이 일어나도 끝나지 않은 트랜잭션때문에 로그의 스페이스가 그렇게 줄지 않는 경우가 있다.

그림 2.3.에서와 같이 트랜잭션이 수행하는 동안 점검점이 일어나도, 트랜잭션 T1같이 오랫동안 실행을 하는 트랜잭션때문에 방화벽이 원하는 만큼 높아지지 않게 된다. 이에 대한 해결책으로 다음과 같이 두가지를 생각할 수가 있다.

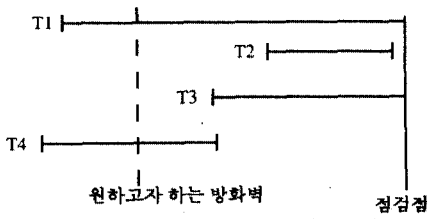


그림 2.3. 점검점이 일어났을때 발생하는 문제

첫번째 방법은 원하는 방화벽 이전부터 실행되고 있는 트랜잭션 프로세스를 강제로 그만두게 하는 것이고, 두번째 방법은 로그를 두개를 두어서 Undo 로그 화일과 Redo 로그 화일을 두어, Redo 로그 화일의 경우 점검점 이후에 이전의 로그 레코드들을 없앨수가 있으므로 로그의 스페이스를 줄일수가 있다. 하지만 이 방법의 경우 Undo 로그 화일의 양은 점검점 이전의 내용때문에

Redo 로그 화일보다 클지 않는다. 그렇기때문에 Undo 로그 화일은 Redo 로그 화일보다 크게 잡아놓는다. COREDB에서는 첫번째 방법을 쓰고 있다.

페이지 로깅방식에서는 로그 레코드를 기록할때는 이전 데이타(before image)와 이후 데이타(after image)를 모두 기록하는 것이 보통이다. COREDB는 연산 로깅을 지원하기때문에, 로그의 양을 줄일수가 있다. 삽입연산인경우는 이후 데이타만 삭제연산인 경우는 이전 데이타만 변경연산인 경우는 모두 필요하게 된다. 이렇게 연산로깅을 하게 되면, 페이지 로깅보다 그 양이 많이 줄게 되고, 데이타의 양을 줄이게 위해서 따로 압축기법등을 쓸 필요가 없다.

## 3.회복기법 모델링

COREDB의 저장구조 파트에서는 1장에서 이야기한 것과 같이 다음과 같이 몇개의 모듈로 이루어진다. 동시성 제어 모듈, 회복 모듈, 저장 모듈, 이렇게 3개의 모듈로 나누어지는데, 모델링을 하기 위해서 이를 세부적으로 더 나눌수가 있다.

여기서 세부적으로 나눈 것들은 프로세스라고 불르기로 했다. 동시성 제어 모듈은 여러개의 트랜잭션 프로세스로 나뉘어진다. 트랜잭션 프로세스는 새로운 트랜잭션이 생기면 새로 만들어진다. 회복기법 모듈에서는 두개의 프로세스로 나뉘게 된다. 로그 화일 관리 프로세스와 점검점 관리 프로세스이다. 점검점 관리 프로세스는 정상 수행시에는 기다리고(wait) 있다가, 점검점이 시작되면 실행을 하는 프로세스이다. 저장모듈에는 하나의 버퍼 관리 프로세스와 여러개의 디스크 관리 프로세스로 나뉘어진다. 디스크 관리 프로세스는 각 화일 블록마다 하나씩 생성된다.

위와 같이 세부적으로 나뉘어진 각 프로세스들간의 상호 연관 그림은 그림 3.1.에 나타나 있다. 여기서 단선으로 표시된 것은 명령어들이 메시지로 전달되는 것이고 복선으로 표시된 것은 데이타 경로이다. 데이타 경로라고 하는 것은 읽거나 쓸때 데이타가 전달되는 것이다.

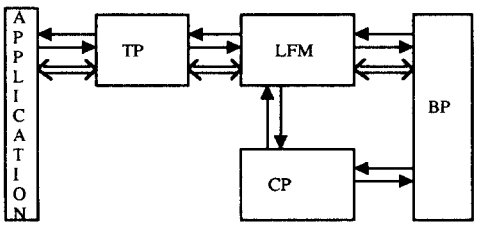


그림 3.1. 프로세스간의 상호 연관도

### 3.1. 트랜잭션 프로세스

트랜잭션 프로세스는 어플리케이션으로부터 연산 명령을 받고, 그 연산 명령을 실행한다. 이 트랜잭션 프로세스에서의 입력은 어플리케이션으로부터와 로그 화일 관리 프로세스로부터 받는다. 출력은 어플리케이션과 로그 화일 관리 프로세스로 보내지게 된다.

트랜잭션 프로세스의 입력을 두가지로 나눌 수가 있다. 어플리케이션으로부터 오는 입력과 로그 화일 관리 프로세스로부터 오는 입력으로 나눌수가 있다.

$\Sigma_{AP,TP}$ = {BOT, READ, WRITE, ABORT, COM, UNDO}
$\Sigma_{LFM,TP}$ = {FAIL, SUC}

이 입력들에 의해서 실제로 실행되는 과정은 그림 3.2.와 같이 나타낼 수가 있다.

### 3.2. 로그 화일 관리 프로세스

로그 화일 관리 프로세스는 로그 화일에 내용을 기록하고, 재실행시 데이타베이스 시스템을 복구시켜주는 프로세스이다. 이 프로세스는 그림 3.1.과 같이 트랜잭션 프로세스와 점검점 프로세스 버퍼 관리 프로세스와 연결되어 있다. 이 로그 화일 관리 프로세스는 3개의 프로세스로부터 입력을 받는다. 각 프로세스별로 받는 입력을 표현하면 다음과 같다.

$\Sigma_{SYSTEM}$ = {restart, shutdown}
$\Sigma_{TP,LFM}$ = {BOT, read, write, undo, commit, abort}
$\Sigma_{BP,LFM}$ = {fail, suc}
$\Sigma_{CP,LFM}$ = {EOC}
$\Sigma_{LFM,LFM}$ = {BOC}

이 입력들에 의해서 실제로 실행되는 과정은 그림 3.3.과 같이 나타낼 수 있다.

### 3.3. 점검점 프로세스

점검점 프로세스는 로그화일 관리 프로세스가, 로그가 어느 정도 찼다고 점검점이 일어나기를 바란다는 메시지를 받으면 수행이 된다. 이 점검점 프로세스는 로그화일 프로세스와는 별도로 수행이 된다. 점검점 프로세스의 수행이 다 끝나면 점검점이 끝났다는 것을 로그에 기록하기 위해서 로그 화일 관리 프로세스에 메시지를 전달하게 된다.

점검점 프로세스로의 입력은 로그화일 관리 프로세스로부터 온 BOC(Begin Of Transaction)밖에 없다.

$\Sigma_{LFM,CP}$ = {BOC}

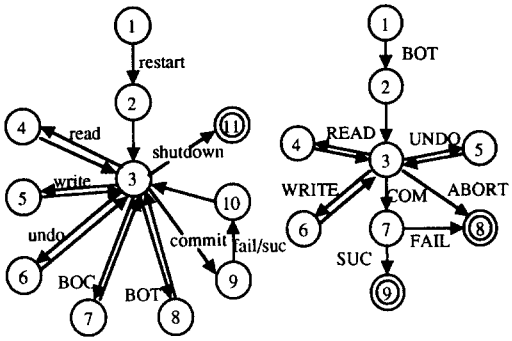


그림 3.2 3.3 state diagrams

## 4. COREDB의 회복 모듈 구현

### 4.1. 데이타 구조

로그 레코드는 모든 정보를 다 가지고 있어야 하고, 자체 로그 레코드에 대한 확인작업정보를 가지고 있어야 한다. 위의 조건에 맞도록 구조를 만들면 다음 과 같은 정보가 필요하게 된다.

| | |
|---|---|
| *LSN* | 로그 레코드의 LSN |
| *kind* | 로그 레코드의 종류 |
| *TransID* | 로그레코드의 트랜잭션 ID |
| *Prev_LSN* | 바로전의 로그 레코드의 LSN |
| *RecordID* | 데이타 레코드 ID |
| *PageID* | 데이타가 페이지 ID |
| *UndoNxtLSN* | 다음번 취소 로그 레코드 LSN |
| *length* | 로그 레코드의 길이 |
| *before_len* | 이전 데이타의 길이 |
| *after_len* | 이후 데이타의 길이 |
| *before_image* | 이전 데이타 |
| *after_image* | 이후 데이타 |

트랜잭션 테이블은 로그 화일을 효율적으로 관리하기 위해서, 그리고 회복기법을 사용하는데 트랜잭션에 관한 정보를 갖고 있

다. 이 화일은 취소나 재실행시 필요한 화일인데, 그렇지 않을 경우 매번 로그를 처음부터 끝까지 살펴보아야 한다.

| | |
|---|---|
| *TransID* | 트랜잭션의 ID |
| *state* | 트랜잭션의 상태 |
| *current_LSN* | 가장 최근에 기록된 로그 레코드의 LSN |
| *UndoNextLSN* | 취소시 읽어야할 로그 레코드의 LSN |
| *FirstLSN* | Begin Transaction 로그 레코드의 LSN |

변경 페이지 테이블은 점검점 기간동안 어떤 페이지들을 디스크에 반영시켜야 하는지에 대한 정보를 갖고 있다. 이 dirty 페이지 테이블에 들어가는 내용은 페이지 업데이트가 일어났을때, 페이지 업데이트에 대한 정보를 기록한다.

이 변경 페이지 테이블에는 다음과 같은 정보가 들어가 있다.

| | |
|---|---|
| *PageID* | 변경된 페이지의 페이지 ID |
| *LSN* | 변경되었을때 기록된 로그 레코드의 LSN |

마스터 영역은 로그 레코드를 기록하거나, 점검점, 재실행시에 필요한 정보를 갖고 있다. 로그 레코드를 기록하는 정상 수행때에는 현재 로그 화일에 포인터가 어디에 있는가하는 정보들을 들 수가 있다.

이 마스터 영역의 구조는 다음과 같다.

| | |
|---|---|
| *Current_LSN* | 쓰여져야 할 로그 레코드의 LSN |
| *first_LSN* | 방화벽의 LSN |
| *Chkpt_LSN* | 가장 최근 점검점의 시작 로그 레코드의 LSN |
| *Stable_LSN* | 디스크에 반영된 마지막 로그 레코드의 LSN |
| *Num_Pages* | 로그 화일의 크기 |
| *Num_Free* | 로그 화일에 남은 크기 |
| *FileID* | 화일 ID |

### 4.2.UNDO

```
UNDO(TransID, SaveLSN)
  UndoNxt = TransTable[TransID].LastLSN;
  WHILE SaveLSN < UndoNxt DO
    LogRec = Log_Read(UndoNxt);
    SELECT(LogRec.kind) DO
      WHEN('update') DO
        IF LogRec is undo-able THEN
          Undo and write CLR;
        UndoNxt = LogRec.PrevLSN;
      END;
      WHEN('compensation')
        UndoNxt = LogRec.UndoNxtLSN;
      OTHERWISE
        UndoNxt = LogRec.PrevLSN;
    END;
  END;
```

### 4.3.RESTART

재실행시에는 3단계로 수행을 한다. 해석을 한 후에 REDO를 하여서 시스템 고장이 일기전상태로 만든다음에 UNDO를 하여 시스템 고장시에 수행중이던 트랜잭션의 효과를 없앤다. 그리고 점검점을 실행하여 데이타베이스를 일관적인 상태로 만든다.

#### 4.3.1.RESTART_ANALYSIS

```
RESTART_ANALYSIS(RedoLSN)
  initialize TransTable, DirtyTable;
  LogRec = next to latest Begin_Chkpoint Record;
  WHILE NOT End of Log DO
    SELECT(LogRec.Type) DO
      WHEN('update' | 'compensation')
        update or undo as LogRec.Type;
      WHEN('End_Chkpt')
        update TransTable & DirtyTable;
      WHEN('prepare')
        TransTable[LogRec.TransID].State = prepare;
      WHEN('abort')
        TransTable[LogRec.TransID].State = abort;
      WHEN('end')
        delete TransTable[LogRec.TransID];
    END;
  RedoLSN = min(entry LSN of Dirtypage);
  END;
```

#### 4.3.2.RESTART_REDO

```
RESTART_REDO(RedoLSN)
  LogRec = next to RedoLSN;
  WHILE NOT End of Log DO
    IF LogRec.Type = 'update' | 'compensation' &
      LogRec.LSN  >=  DirtyTable[LogRec.PageID].RecLSN
THEN
      update;
  END;
```

#### 4.3.3.RESTART_UNDO

```
RESTART_UNDO()
  WHILE EXISTS Trans.State = 'U' in TransTable DO
    LogRec = pick up Log with maximum UndoNxtLSN;
    SELECT(LogRec.Type) DO
      WHEN('update') DO
        undo and write CLR;
        if LogRec.PrevLSN = 0
          write 'end' Log;
      END;
      OTHERWISE
        update trans.UndoNxtLSN;
    END;
  END;
```

### 4.4.CHECKPOINT

```
CHECKPOINT()
  Write 'Begin_Checkpoint';
  save TransTable and DirtyTable;
  for each entry in saved DirtyTable
    flush Pages and clear DirtyTable;
  Write  'End_Chkpoint'  with  saved  TransTable  &
DirtyTable;
  flush Master;
```

## 5.결론

본논문에서 다룬 회복기법은 안정성과 속도향상에 중점을 두었다. 그리고 구현상에서 되도록 본논문에서 다루었던 문제점과 그 해결방법을 이용하여 현재 COREDB의 회복 모듈을 만들고 있다.

앞으로 할일은 레코드 단위 잠금과 한 페이지가 넘는 긴 레코드를 지원하는 회복기법을 만들 것이다. 그리고 현재는 생각지 않은 메디아 고장을 위한 백업저장을 시스템의 정지없이 수행할 수 있는 장치도 할 생각이다.

### 참고문헌


[1] Theo Haerder and Andreas Reuter, "Principles of Transaction - Oriented Database Recovery", *Computing Surveys, Vol. 15, No. 4, December 1983*

[2] Y.C. Kim, et al. "A Hypercube Database Computer - COREDB", *submitted int the 1992 Phoenix Conference on Computers and Communications*

[3] Raymond A. Lorie, "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977, Pages 91-104*

[4] Jim Gray, Paul McJones, et al. "The Recovery Manager of the System R Database Manager", *Computing Surveys, Vol. 13, No. 2, June 1981*

[5] C. Mohan, et al. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *IBM Research Report, 1989*