클라이언트-서버 DBMS 로의 ARIES 확장[†]
(Extending ARIES for Client-Server DBMS)

서울대학교 계산통계학과 전산과학 전공 이강우
서울대학교 컴퓨터공학과 김형주

요약

저렴하고 강력한 워크스테이션과 고속의 네트워크의 등장으로, 과학 분야 그리고 공학 분야에서 클라이언트-서버 구조는 공유 자원에 접근하는 매우 일반적인 계산 환경으로 자리잡게 되었다. 이러한 결과로, 클라이언트-서버 DBMS 구조는 많은 사람들에 의해서 연구되고 있다. 그러나 클라이언트-서버 DBMS 의 고장회복에 관한 연구는 아직까지 깊이 있게 연구되지 않고 있는 실정이다. 본 논문은 현재 고장회복 기법으로 많은 관심을 받고 있는 ARIES 고장회복 알고리즘을 클라이언트-서버 구조 DBMS 로 확장시켰을 때, 발생 가능한 문제점에 대한 해결방법을 제시하고, 기존의 방법과의 비교를 통해 우수성을 보였다.

Abstract

With powerful workstations and fast networks becoming commonplace in scientific, engineering environment, *client-server* architectures have become popular appoaches to providing access to shared services and resources. As a result, many researchers have studied, namely, the client-server DBMS architecture. But crash recovery in client-server DBMS is not yet understood well. In this paper, we provide the solutions to the problems which caused by accommodating the recovery algorithm, ARIES, to client-server DBMS architecture. And we also compare our results to a previous work.

[†]이 연구는 '92년도 한국과학재단 연구비지원에 의한 결과임 (과제번호:921-1100-003-2).

1 연구 동기 및 필요성

컴퓨터 하드웨어와 통신 장비의 급속한 기능 향상과 가격의 저렴화로 인하여, 데이타베이스 관리 시스템(DataBase Management System: DBMS)의 구조는 많은 영향을 받게 되었다. 기존의 데이타베이스 관리 시스템은 주로 메인프레임이나 대형 컴퓨터와 같은 강력한 컴퓨터에 설치되고, 이 컴퓨터에 터미날을 연결하여 운용하는 방식이 대부분이었다. 그러나 워크스테이션과 같이 비교적 저렴하고 중/대형 컴퓨터의 기능을 능가할 정도의 기능을 갖춘 컴퓨터의 개발과, LAN과 같이 고속으로 대용량의 데이타를 전송시킬 수 있는 통신 장비의 개발로 인하여, 클라이언트-서버 계산 환경이 이제는 과학 분야, 공학 분야, 또한 사무 자동화 분야에서도 일반적인 환경이 되었다. 이에따라 클라이언트-서버 환경을 이용한 클라이언트-서버 DBMS 가 많은 사람들로부터 큰 호응을 받기 시작하였다. 이러한 추세는 DBMS 구조에도 많은 영향을 미쳐, 클라이언트-서버 DBMS 는 많은 연구자로부터 연구의 대상이 되었다. 더구나 클라이언트-서버 DBMS는 CASE 정보 저장소(CASE repository), CAD/CAM, CAE, 사무 자동화, 그리고 인공지능 분야와 같이 최근에 새롭게 등장한 DBMS 어플리케이션에 매우 적합한 프래폼(platform)으로 평가받고 있다.

일반적으로 클라이언트-서버 DBMS는 데이타베이스와 DBMS를 하나의 컴퓨터(서버라 부름)에 적재시키고, 기타 다른 컴퓨터(클라이언트라 부름)들을 고속의 네트워크를 이용하여 서버에 연결하여 클라이언트에서 사용자가 데이타를 서버에게 요청하고, 서버로부터 요청한 데이타가 도착하면 이것을 처리하여 사용자에게 보여주는 방법으로 질의를 처리하게 된다(그림 1 참조).

그리고 일반적으로 클라이언트-서버 DBMS에서는 데이타를 갱신하는 트랜잭션은 클라이언트에서 수행되고, 고장회복시에 수행되는 고장회복 작업은 서버에서 수행된다. 그러므로 클라이언트-서버 구조에서의 고장회복 알고리즘은 기존은 중앙집중식 DBMS에서의 그것보다 복잡하게 된다[1]. 그러나 지금까지 발표된 논문에 아직까지 이 분야에 대한 깊은 연구가 되고 있지 않은 실정이다. 특히 복잡한 고장회복 알고리즘을 요구하는 부분 철회와 레코드 잠금단위와 같은 페이지 잠금단위보다 작은 잠금단위를 지원하는 고장회복 알고리즘은 중앙집중식 DBMS에서는 어느 정도 연구되었으나, 이러한 고장회복 알고리즘을 클라이언트-서버 DBMS로 확장시키는 연구는 아직까지 미비하다.

본 논문은 기존의 고장회복 알고리즘을 클라이언트-서버 환경으로 확장시키는 방법을 제안하고, 우리가 제안한 방법과 기존의 방법을 비교하였다.

본 논문의 구성은 다음과 같다. 먼저 2 절에서는 고장회복 기법에서 사용되는 여러가지 용어와 기본적인 개념에 대해 간략하게 설명하고, 3 절에서는 현재 고장회복 알고리즘으로 가장 많이 구현되고 있는 ARIES에 대해 간단하게 설명하고, 4 절에서는 기존에 연구된 클라이언트-서버 DBMS 환경하에서의 고장기법에 대해 설명하고, 본 논문에서 제안한 고장회복 기법을 기술

[1] 연산 로깅을 사용하지 않은 클라이언트-서버 DBMS에서의 고장회복 알고리즘은 기존의 중앙집중식 DBMS에서의 그것과 거의 비슷할 수도 있다.

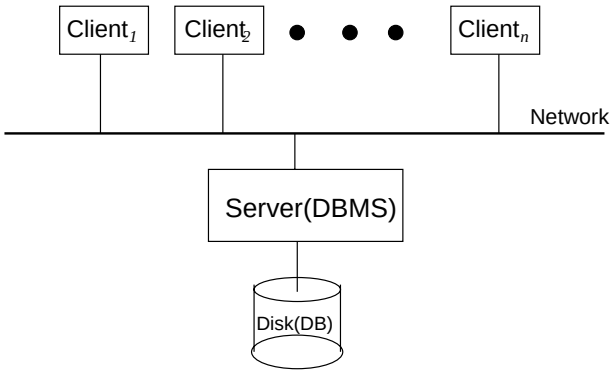

그림 1: 클라이언트-서버 모델

한다. 그리고 본 논문에서 제안한 기법이 기존에 연구된 방법과의 차이를 비교하고, 이것이 갖는 장점을 설명한다. 마지막으로, 5 절에서는 본 논문에서 도출한 결과를 정리하고, 앞으로 이루어져야 할 연구방향을 제시한다.

2 고장회복 기법의 개요

트랜잭션[1, 9]은 DBMS, 분산 시스템, 그리고 기타 트랜잭션 처리 시스템에서 이미 오래 전부터 잘 알려진 개념이다. 트랜잭션은 데이타의 신뢰도가 매우 중요하고, 여러 사람이 동시에 데이타를 접근하는 시스템에서 사용되는 작업의 단위가 된다. 사용자가 어떠한 작업을 처리하려 할 때, 사용자는 하고자 하는 작업을 하나의 트랜잭션화하여 그것을 DBMS 로 하여금 처리하게 한다.

DBMS에서 고장회복 기능을 맡는 모듈을 고장회복 관리자라 부른다. 시스템에 고장이 발생했을 때 일반적으로 고장회복 관리자가 파괴된 일관성을 복구하는 방법은 고장이 발생하던 순간에 아직 수행을 마치지 않은 트랜잭션이 있으면, 그 트랜잭션이 변경시킨 데이타를 트랜잭션 수행 이전의 값으로 되돌려 놓는다(우리는 이러한 작업을 트랜잭션 undo라고 한다). 트랜잭션이 시스템 고장 이전에 그 수행이 완료(commit)되었지만 그것이 변경한 데이타가 미처 디스크에 기록되지 않은 상태에서 고장이 발생하였을 때는 기록되지 못한 데이타를 다시 기록한다(우리는 이것을 트랜잭션 redo라고 한다).

고장회복 관리자가 트랜잭션 undo와 redo를 이용하여 고장회복을 수행하기 위해서는 고장회복 관리자가 트랜잭션이 정상작동 중에 어떻게 데이타를 변경시켰는가를 알아야 한다. 그러므로 DBMS 는 데이타에 변경이 발생될 때마다 그 사실을 일일이 로그(log)라는 특정 화일에 기록한다. 그러므로 시스템에 고장이 발생하였을 때, 고장회복 관리자는 이 화일에서 필요한 정보를 얻어 시스템 고장 당시의 트랜잭션 상태에 따라 적합한 복구를 하게된다. 위와같은 방법으로 시스템의 고장을 복구하는 방법을 로깅(logging)[2][1, 7]이라 부른다.

로깅을 사용하는 대부분의 시스템은 로그 레코드를 로그에 기록할 때, 각 로그 레코드마다 하

[2]로깅을 이용한 고장회복 외에 새도우(shadow) 페이지를 이용한 고장회복 방법[1, 6, 9]도 있으나 상용화된 대부분의 DBMS는 로깅을 이용한 고장회복 방법을 사용한다.

나의 로그 순차 수(Log Sequence Number: LSN)를 부여한다. LSN은 모든 로그 레코드에 대해 유일하게 정의된다. 로그 레코드는 고장회복 관리자가 고장 회복시에 필요한 undo 내지는 redo를 할 수 있는 충분한 데이타가 저장되어야 한다. Undo와 redo를 위한 정보가 동시에 기록된 로그 레코드를 undo-redo 로그 레코드라 부르고, undo(redo)를 위한 정보만 있는 로그 레코드를 undo 전용(redo 전용) 로그 레코드라 부른다.

로그 레코드의 redo/undo 정보가 저장된 형식에 따라 로그 레코드는 연산 로그 레코드(operational log record)와 물리적 로그 레코드로 구분될 수 있다. 물리적 로그 레코드는 redo/undo 정보가 기록되는 방식이 갱신이 발생하였을 때, 어느 페이지의 어느 부분이 어떻게 변화되었는가를 기록한 것이고, 연산 로그 레코드는 어느 페이지의 어느 부분에 어떠한 연산이 적용되었는지가 기록된 것이다. 연산 로그 레코드가 물리적 로그 레코드보다 기록되는 양이 적어 매우 유용하나, 고장회복 과정에서 필요한 idempotent 성질을 구현하기 어렵다.

로그를 사용하여 고장회복을 수행하는 시스템은 버퍼에 있는 데이타 페이지를 디스크에 쓸 때, *Write-Ahead-Log(WAL)* 규약을 준수한다. WAL 규약이란, 버퍼에 있는 데이타 페이지를 디스크에 쓸 때, 그 데이타 페이지를 변경시킴으로써 생성된 모든 로그 레코드를 먼저 로그 화일에 저장한 후, 데이타 페이지를 디스크에 저장하는 규약을 말한다. 로깅을 이용한 고장회복을 위해서는 한 트랜잭션이 그 수행을 완료하기 전에 반드시 그것이 발생시킨 모든 로그 레코드를 반드시 안전 디스크에 저장하여야 한다.

로그를 사용하여 고장회복을 구현하는 대부분의 시스템의 경우, 트랜잭션 철회시 혹은 재 가동 undo시, undo 를 수행한 사실을 로그 레코드를 이용하여 로그에 기록한다. 이러한 로그 레코드는 상쇄 로그 레코드(Compensate Log Record: CLR)[2, 12]라 한다.

3 ARIES 고장회복 기법

ARIES 는 고장회복 알고리즘으로서, 지금까지 여러 DBMS 의 고장회복 기법으로 구현되었다. 본 논문에서 ARIES 를 이용하여 클라이언트-서버 DBMS 의 고장회복 기법을 설계하고자 하는 이유는 ARIES 의 강력함과 단순함에 있다. 이 절에서는 ARIES 에 대한 간단한 설명을 한다. 본 논문에서 설명하는 ARIES 에 대한 자세한 이해을 얻고자 하는 독자는 [12]를 참조하기 바란다.

3.1 ARIES 를 위한 데이타 구조

3.1.1 로그 레코드

로그 레코드의 종류에는 데이타를 갱신할 때 생성되는 갱신 로그 레코드, 철회할 때나 undo 할 때 발생되는 상쇄 로그 레코드, 그리고 트랜잭션이 종료할 때 생성되는 로그 레코드 등이 있다. 로그 레코드의 구조는 그 종류에 따라 다를 수 있는데, 이들 중 중요한 필드를 설명하면 다음과 같다.

LSN: 로그 레코드의 LSN.

Type: 로그 레코드의 종류를 기록하는 필드. 예를들어 이 필드에는 갱신 로그 레코드를 표시하기 위한 '*update*', 상쇄 로그 레코드를 위한 '*compensate*' 가 있다.

TransID: 로그 레코드를 생성한 트랜잭션의 식별자.

PrevLSN: 같은 트랜잭션에 의해서 바로 이전의 생성된 로그 레코드의 LSN. 이 필드는 트랜잭션 undo를 수행할 때 사용된다.

PageID: 갱신이 적용된 디스크 데이타 페이지의 식별자.

UndoNxtLSN: 상쇄 로그 레코드일 때만 필요한 필드로서, 어느 로그 레코드부터 철회를 시작해야 하는지를 알려주는 LSN. 갱신 효과가 undo된 로그 레코드의 PrevLSN 필드 값을 갖게 된다(3.2.2 절 참조).

Data: Undo/redo를 위한 정보가 기록.

3.1.2 페이지 구조

모든 데이타 페이지에 공통적으로 포함되는 필드로 pageLSN 이 있는데, 여기에 가장 최근에 페이지를 갱신하여 생긴 로그 레코드의 LSN 이 기록된다. 이 필드가 필요한 이유는 첫째, 버퍼 관리자가 WAL 규약을 구현할 때 필요하고, 둘째, pageLSN을 이용하여, 로그 레코드가 해당 페이지에 적용 되었는지 여부를 판가름 할 수 있어, pageLSN 은 연산 로깅을 지원하는 시스템에서 idempotent 한 고장회복 구현에 유용하게 사용된다.

3.1.3 트랜잭션 테이블

시스템에 고장이 발생한 후, 재 가동 될 때 고장회복 관리자가 트랜잭션의 상태를 추적하기 위해 사용하는 테이블이다. 이 테이블은 시스템 고장 후, 재 가동될 때 최근의 검사점 로그 레코드(*checkpoint log record*)에 기록된 트랜잭션 테이블로 초기화된 후, 검사점 로그 레코드로부터 로그 레코드를 로그 레코드가 생성된 시각 순서대로 차례로 읽으면서 만일 트랜잭션의 완료 로그 레코드내지는 실패 로그 레코드가 발견되면 트랜잭션 테이블에서 해당 트랜잭션을 제거하고, 만일 트랜잭션의 시작 로그 레코드가 발견되면 그 트랜잭션을 트랜잭션 테이블에 등록하는 방법으로 트랜잭션의 상태를 추적하게 된다. 이 트랜잭션 테이블은 정상 수행 상태에서도 트랜잭션 관리자에 의해서 관리된다.

효과적인 트랜잭션 상태 추적을 위한 트랜잭션 테이블의 중요한 필드로, 트랜잭션 식별자를 기록하는 *TransID*, 트랜잭션에 의해 발생된 로그 레코드 중에서 가장 최근 것의 LSN 을 기록하는 *LastLSN*, 트랜잭션 undo 할 때 처음으로 사용되어야 할 로그 레코드의 LSN 을 기록하는 *UndoNxtLSN*이 있다.

3.1.4 Dirty 페이지 테이블

Dirty 페이지 테이블[3]은 DBMS가 정상적으로 수행될 때, 버퍼 관리자에 의해서 dirty된 페이지의 상태를 기록하기 위해 사용되고(버퍼 풀 DPT이라 부름), 재가동 고장 회복시에도 사용된다(재가동 DPT라 부름). 어떤 버퍼 페이지가 dirty되었다 함은 페이지가 디스크로부터 읽혀져 버퍼에 올라와, 트랜잭션 혹은 임의의 프로세스에 의해 그 내용이 변경된 후, 그 변경 내용이 아직 디스크 페이지로 쓰여지지 않음을 의미한다. DPT 의 필드로 페이지 식별자를 기록하는 *PageID*, 페이지를 처음으로 dirty로 만든 갱신에 의해 생성된 로그 레코드의 LSN 을 기록하는 *RecLSN* 이 있다.

3.2 정상 작동

3.2.1 갱신 작동

ARIES를 고장회복 알고리즘으로 사용하는 시스템에서는 트랜잭션이 한 데이타 페이지에 속해 있는 레코드를 갱신할 때 먼저 레코드를 갱신한 후, 로그 레코드를 하나 만들어, 그것의 Type 을 '*update*'로 기록하고, TransID 는 트랜잭션의 식별자로 기록하고, Data 필드는 갱신 내용에 따른 redo/undo 정보를 기록하고, PageID 에 갱신이 적용된 페이지의 식별자를 기록한다.

TransID 를 이용하여 트랜잭션 테이블을 검색하여 해당 항목이 없으면 트랜잭션을 테이블에 등록하고, 등록한 항목의 LastLSN과 UndoNxtLSN을 생성된 로그 레코드의 LSN으로 기록하고, 로그 레코드의 PrevLSN은 0로 기록한다 (PrevLSN이 0인 로그 레코드는 트랜잭션에 의해 첫번째로 생성된 로그 레코드임을 표시하게 된다). 만일 트랜잭션 테이블 검색에 의해 해당 항목이 검색되면, 생성된 로그 레코드의 PrevLSN 에 검색된 항목의 LastLSN 을 기록하고, 항목의 LastLSN 과 UndoNxtLSN 에는 로그 레코드의 LSN 을 기록한다.

페이지 갱신이 버퍼에 올려진 페이지의 첫번째 갱신인 경우에는 갱신된 PageID 를 이용하여 버퍼 풀 DPT 에 등록하고, 등록된 항목의 RecLSN을 생성된 로그 레코드의 LSN 으로 기록한다. 마지막으로 페이지 내의 pageLSN을 생성된 로그 레코드의 LSN 으로 기록한다.

3.2.2 부분 철회 및 전체 철회

트랜잭션이 철회될 때는 먼저, 트랜잭션의 식별자를 이용하여 트랜잭션 테이블에서 해당 항목를 찾아 항목이 검색되면 그것의 UndoNxtLSN에 해당하는 로그 레코드부터 undo를 시작하여, PrevLSN (또는 UndoNxtLSN)으로 연결된 로그 레코드를 따라가면서 트랜잭션 철회를 수행한다. Undo할 로그 레코드를 읽어 로그 레코드의 Type이 update라면, 그것의 PageID에 해당하는 페이지를 읽은 후, 로그 레코드의 Data 필드에 있는 정보를 이용하여 undo 한뒤, 그 사실을 상쇄 로그 레코드를 이용하여 로깅하되, 상쇄 로그 레코드의 UndoNxtLSN 에 undo 에 사용한 로그 레코드의 PrevLSN 을 기록한다. 트랜잭션 테이블의 LastLSN 필드에는 상쇄 로그 레코드의 LSN 을 기록하고, 데이블의 UndoNxtLSN 필드에는 undo에 사용된 로그 레코드의 PrevLSN 을

[3] 편의상 앞으로 DPT라 약칭하여 사용한다.

기록한다. 다음에는 로그 레코드의 PrevLSN 를 이용하여 다음으로 undo 할 로그 레코드를 접근한다.

트랜잭션 철회 중 undo 에 사용될 로그 레코드의 Type 이 compensate 라면, 이 로그 레코드를 이용하여 undo 를 수행하지 않고, 단지 다음으로 undo 시킬 로그 레코드를 상쇄 로그 레코드의 UndoNxtLSN 을 이용하여 접근한다(갱신 로그 레코드의 경우에는 다음으로 undo 할 로그 레코드를 PrevLSN 으로 찾음에 유의하라.).

트랜잭션 철회작업은 로그 레코드의 PrevLSN (상쇄 로그 레코드의 경우에는 UndoNxtLSN)이 0일 때까지 계속되거나(전체 철회인 경우), 로그 레코드의 PrevLSN(상쇄 로그 레코드인 경우에는 UndoNxtLSN)에 해당하는 로그 레코드가 안전점 로그 레코드일 때까지(부분 철회인 경우) 계속된다.

3.3 재 가동 고장회복(Restart Recovery)

ARIES 를 고장회복 알고리즘으로 사용하는 시스템은 시스템 고장이 발생하여 재 가동될 때 분석 단계, redo 단계, 그리고 undo 단계를 거치는 재 가동 고장회복 단계를 수행한다. 분석 단계는 최근의 검사점 로그 레코드부터 시작된다. 분석 단계를 통해서 얻어진 결과를 이용하여, redo 단계, undo 단계를 수행한다. 이 절에서는 지면 관계로 재 가동 고장회복시 수행되는 작업을 매우 간략하게 설명한다. 관심이 있는 독자는 [12]를 참조하기 바란다.

3.3.1 분석 단계(Analysis Pass)

DBMS가 시스템 고장으로 부터 재 가동될 때, 고장회복 관리자는 처음으로 시스템 실행 제어를 획득하여 재 가동 고장회복을 실시한다. 먼저 고장회복 관리자는 안전 디스크의 마스터 레코드로부터 최근의 검사점 로그 레코드를 읽어, 이 로그 레코드부터 분석 단계를 시작한다. 분석 단계에서는 최근의 검사점으로부터 시작하여, 시스템 고장이 발생하기 바로 전까지 로그 레코드가 생성된 시각 순서로 로그 레코드를 차례로 읽으면서 트랜잭션의 상태와 dirty 페이지의 상태를 계속 추적하게 된다. 이러한 추적을 통하여 다음으로 수행될 redo 단계가 어느 로그 레코드부터 시작되어야 하는지를 분석하고, undo 단계에서 어느 트랜잭션들을 철회시킬지를 가려내게 된다.

3.3.2 Redo 단계(Redo Pass)

Redo 단계는 분석 단계로부터 건내 받은 정보를 이용하여 해당 로그 레코드부터 로그 레코드를 하나하나 차례대로 읽으면서 redo 가 시작된다. Redo 단계에서 주의깊게 볼 점은 트랜잭션의 최종 결과가 완료되었는지 아닌지와는 관계없이 redo 한다는 사실이다. 즉, 모든 로그 레코드의 갱신 효과를 다시 그대로 데이타베이스에 반영시킨다. 이러한 방법으로 시스템 고장 바로 이전의 로그 레코드까지 redo 하면(즉, redo 단계가 끝나면), 시스템 전체적인 상태는 시스템 고장 바로

직전의 상태가 된다. 이러한 redo 는 기존의 고장회복 방법에서의 redo[4]와는 달리 매우 간단하면서도 강력한 고장회복 알고리즘을 제공한다([12] 참조).

3.3.3 Undo 단계(Undo Pass)

Redo 단계에서 트랜잭션의 완료 여부와 관계없이 redo 가 수행되었기 때문에, undo 단계를 시작할 때의 DBMS 의 상태는 시스템 고장 바로 직전의 상태가 된다. 그러므로 undo 단계 수행 중에는 실제로 갱신 효과가 페이지 반영이 되었는지 아닌지를 판단할 필요가 없다. 이 상태에서의 undo 단계는 시스템 고장 이전의 상태에서 완료되는 않은 모든 트랜잭션들을 전체 철회시키는 것과 차이가 없게 되어, undo 단계가 매우 간단하게 된다. 더구나 redo 단계에서 상쇄 로그 레코드에 대한 redo 가 다시 되기 때문에 (즉, 갱신에 대한 undo 가 이미 수행되었기 때문에), 이전에 undo 되었던 페이지 중에서 그 undo 효과가 시스템 고장으로 인해 없어졌는지 여부로 판단할 필요가 없게 되고, 이러한 상쇄 로그 레코드는 다음으로 undo 해야할 로그 레코드를 직접 가리키기 때문에 매우 효율적으로 undo 가 수행된다.

앞서 말한 바와같이 undo 는 고장 바로 직전에 완료되지 않은 모든 트랜잭션들을 전체 철회하는 작업과 거의 동일하여, 앞서 시스템이 정상 작동할 때의 트랜잭션 전체 철회 알고리즘을 그대로 사용된다. 다만 트랜잭션 전체 철회에서는 하나의 트랜잭션만을 철회하기 때문에 재 가동 undo 단계에 약간의 수정이 필요한데, 그것은 다음으로 undo 에 사용할 로그 레코드를 찾는 방법이다. 그 방법은 트랜잭션 테이블에 있는 모든 항목의 UndoNxtLSN 필드들 중에서 가장 큰 LSN 에 해당하는 로그 레코드를 다음으로 undo 시킬 로그 레코드로 삼는 것이다. 이 로그 레코드를 이용하여 undo 가 수행되면, 그 로그 레코드의 PrevLSN (혹은 UndoNxtLSN)을 트랜잭션 테이블 내 해당 항목의 UndoNxtLSN 에 기록한 뒤, 다시 전체 트랜잭션 테이블 내의 UndoNxtLSN 중에서 가장 큰 LSN 을 찾는 방법을 사용하여, 다음으로 undo 할 로그 레코드를 찾는다. 위와같은 방법을 계속 반복하여, 트랜잭션 테이블 내의 모든 항목의 UndoNxtLSN 이 0가 되면 undo 단계가 끝나게 된다.

4 ARIES 의 클라이언트-서버 DBMS 를 위한 ARIES 의 확장

4.1 클라이언트-서버 모델(Client-Server Architecture)

클라이언트가 사용자로부터 질의어를 받아 직접 해석하여, 필요한 데이타를 서버에게 직접 요구하여 질의를 처리하는 구조를 데이타 전송(data-shipping) 클라이언트-서버 구조라 한다[3]. 데이타 전송 구조는 객체지향 DBMS 나 최근에 개발된 저장 관리자에서 주로 사용되고 있는 구조이다. 클라이언트와 서버 간의 전송되는 데이타의 기본 단위가 페이지인 데이타 전송 구조의 서버를 페이지 서버라 부르는데, 페이지 서버 구조를 따르는 시스템으로는 클라이언트-서버 EXODUS[4, 5], ObjectStore[10], ObServer[8] 등을 들 수 있다.

[4] 예를들면 System R, SQL/DS, DB2에서의 고장회복 방법은 완료된 트랜잭션에 의해 반영된 갱신효과만을 redo 한다.

본 논문에서는 페이지 서버를 가정한다. 그리고 클라이언트는 일정 양의 버퍼 풀을 갖고 있고, 이것을 자체적으로 관리한다고 가정하고, 연산 로깅을 수행하는 DBMS 를 가정한다. 그리고 클라이언트와 서버에서 관리되는 버퍼 풀의 각 버퍼 슬롯마다 버퍼 제어 블럭(buffer control block)[12]을 두어, 버퍼 관리에 필요한 여러가지 정보를 기록한다고 가정한다. 예를들어 버퍼 제어 블럭에 기록될 수 있는 정보로는 버퍼 슬롯의 dirty 여부를 기록하는 dirty 프래그, 버퍼 슬롯의 고정(fix) 여부를 기록하는 고정 계수(fix counter), 그리고 래치(latch) 등을 들 수 있다.

이 절에서는 3 절에서 설명한 ARIES 고장회복 기법을 클라이언트-서버 DBMS로 어떻게 확장시킬 것인가에 대한 설명을 한다. 기존의 ARIES는 중앙집중식 DBMS를 가정한 기법이므로, ARIES를 그대로 클라이언트-서버 DBMS의 고장회복 알고리즘으로 사용하기에는 문제가 있다. ARIES 고장회복 기법을 데이타 전송 클라이언트-서버 모델에서 직접 사용할 수 없는 주요 원인은 다음과 같다[5].

- 데이타의 생성, 갱신, 제거와 같은 데이타를 변경시키는 연산과 로그 레코드 생성은 클라이언트에서 수행되는 반면, 데이타와 로그 레코드는 서버에 저장된다.
- 클라이언트와 서버 사이의 데이타 전송의 기본 단위는 페이지이다.
- 클라이언트 쪽에서 변경되는 데이타와 이와 관련된 로그 레코드들이 동시에 서버로 전송되지 않을 수 있기 때문에, 로그 레코드가 서버에 도착하는 시각과 데이타 페이지가 도착하는 시각에는 차이가 있다. 그러므로 서버에서는 데이타 갱신에 대한 로그 레코드는 이미 도착하였으나, 갱신된 데이타가 아직 서버의 버퍼 풀에 없을 수 있기 때문[5]에, ARIES를 그대로 사용할 수 없다.

4.2 관련 연구: EXODUS

클라이언트-서버 DBMS 에서의 고장회복 기법에 대한 연구 문헌을 아직까지는 쉽게 접할 수 없기 때문에, 다른 클라이언트-서버 DBMS 에서의 고장회복 기법과 비교는 어려운 상태이다. 다만 Wisconsin 대학에서 개발한 클라이언트-서버 EXODUS 저장 관리자(*Client-Server EXODUS Storage Manager: ESM-CS*)에서 사용되는 고장회복 방법[5][6]과 비교할 수 있다. 이 절에서는 ESM-CS 에서 사용하는 고장회복 기법에 대해서 간략하게 설명한다.

4.2.1 클라이언트에서의 WAL 규약의 구현

ESM-CS 에서는 클라이언트에서 서버로 데이타 페이지 전송할 때 WAL 규약을 따르게 한다. 즉, 클라이언트에서 dirty 페이지를 서버로 전송할 때 그 페이지를 변경시켜 생성된 모든 로그 레코드들을 먼저 (혹은 전송시키려는 데이타 페이지와 함께) 서버로 전송한다. 실제로 클라이언트

[5] 중앙집중식 DBMS의 경우에는 데이타 갱신에 대한 로그 레코드가 있다는 것은 이미 그것에 대한 갱신 효과가 최소한 버퍼에 있는 페이지에 적용되었다는 것을 의미한다.

[6] 참고문헌 [5]에서 제안한 방법을 ESM-CS 에서 구현하였기 때문에, 이후로는 "ESM-CS 에서의 방법" 이란 표현과 "[5]에서 제안한 방법" 이라는 표현을 혼용하여 사용한다.

에서의 WAL 규약이 시스템의 일관성 유지와는 아무런 관계는 없으나, 시스템의 설계 및 구현의 편의상 이러한 규약을 설정한다.

4.1 절에서 설명한 바와 같이 클라이언트와 서버는 각자의 버퍼 풀을 갖고 있어 자체적으로 관리된다고 가정한다. 다만, 클라이언트에서의 버퍼 관리 중, 페이지 교체가 발생되면 버퍼 슬롯의 내용이 클라이언트 컴퓨터의 디스크에 저장되는 것이 아니고, 서버로 전송된다.

클라이언트의 버퍼 관리자도 WAL 규약을 준수해야 하기 때문에, 로그 레코드를 생성시킬 때, 생성된 로그 레코드의 LSN 을 갱신된 페이지의 pageLSN 에 기록할 필요가 있다. 그러나 로그 레코드가 저장될 로그 화일은 서버에 있고, 서버는 여러 클라이언트로부터 생성된 로그 레코드를 저장하기 때문에, [11]에서 제안한 방법과 같이 서버와의 많은 횟수의 메세지 전송이 필요한 방법없이는 클라이언트에서는 생성된 로그 레코드의 LSN 을 알 수 없다. 그러므로 클라이언트에서는 기존의 ARIES 와 같이 LSN 을 이용한 WAL 규약을 구현할 수 없게 된다.

이를 해결하기 위해서 LSN 이 지니는 성질 중 WAL 규약을 구현하는데 필요한 성질만을 뽑아서 클라이언트에서 사용하게 되는데, 이것을 *LRC(Log Record Number)* 라 부른다. LRC 는 한 페이지에 해당하는 모든 레코드에 대해 유일하고, 단조증가하는 성질을 갖기 때문에 클라이언트에서 갱신이 발생할 때 생성되는 로그 레코드의 LRC 를 데이타 페이지에 기록하여(LRC 는 데이타 페이지의 pageLRC 에 기록된다), 한 로그 레코드가 클라이언트에서 서버로 전송될 때, 그 로그 레코드의 LRC 와 클라이어트의 데이타 페이지의 pageLRC 를 비교하여 로그 레코드의 LRC 보다 작거나 같은 pageLRC 를 갖는 데이타 페이지를 서버로 함께 전송하는 방법으로 구현하게 된다. 그리고 서버에서도 로그 레코드에 대한 갱신효과가 해당 데이타 페이지에 적용되었는가를 확인할 때 LRC 가 사용된다. 이를 위해서 로그 레코드에 LRC 를 위한 필드가 추가된다. 그리고 LRC 는 로그 레코드의 실제 위치를 가리키는 물리적 주소가 아니라, 논리적 의미를 갖는 값이기 때문에 클라이언트에서도 효과적으로 관리/사용된다. 고장회복을 효과적으로 하기 위해서는 각 로그 레코드의 물리적 주소를 의미하는 LSN 이 필요하기 때문에, 서버에서는 계속적으로 LSN 이 관리되어야 한다. 즉, LRC 는 기존 ARIES 에서 LSN 이 제공하던 기능 중에서 특정 로그 레코드가 데이타 페이지에 적용되었는가를 조사하는 일을 대신 맡고, LSN 은 단순히 서버에서 주어진 LSN 을 이용하여 해당 로그 레코드를 접근하는데 사용된다.

ESM-CS 는 올바른 LRC 생성을 위해, 서버에서 클라이언트로 데이타 페이지를 전송할 때, *end-of-LSN* 을 함께 전송하여, 클라이언트가 이것을 이용하여 LRC 를 생성하도록 한다.

4.2.2 조건부 Undo(Conditional Undo)

앞서 설명한 바와같이 클라이언트-서버 환경에서는 한 로그 레코드가 서버에 있는다는 사실이 그 로그 레코드로 변경된 데이타가 서버에 반영되었다는 사실을 의미하지 않는다. 그러므로 트랜잭션 철회시 기존의 ARIES에서 사용되었던 무조건[7] 철회는 시스템의 일관성을 파괴시킬 수 있다[5].

[7] 여기서 '무조건'은 undo 를 수행할 때, 페이지의 pageLSN 와 로그 레코드의 LSN 을 비교하지 않고 undo 를 수행한다는 의미이다.

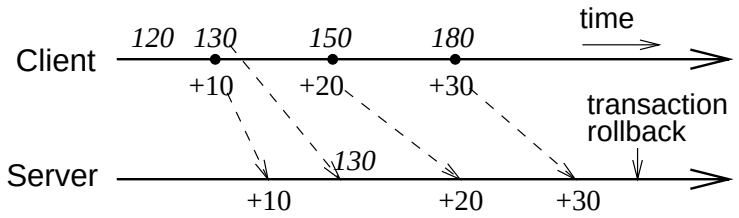

그림 2: 무조건 undo 의 오류의 예

그림 2의 예를보자. 그림 2에서 120은 서버로부터 받은 레코드의 값이고, 이 레코드에 대한 세번의 갱신으로 그것의 값이 130, 150, 180 으로 변경되는 것을 보여주고, 화살표 밑의 수 +10, +20, +30은 갱신에 해당하는 연산 로그 레코드를 보여준다. 또한 이 그림은 세번의 갱신으로 생성된 로그 레코드는 모두 서버에 도착하였고, 첫번째 갱신으로 인해 변경된 데이타 페이지만이 도착한 상태를 보여준다. 이 상태에서 세번째 갱신까지 반영된 데이타 페이지가 아직 서버에 도착하지 않은 상태에서 트랜잭션 철회가 서버에서 발생하였다고 가정하자. 이때 기존 ARIES의 undo 알고리즘은 데이타 페이지의 상태는 체크하지 않은 상태에서 오직 로그 레코드만을 이용하여 undo 를 수행하기 때문에 3.2.2 절에서 설명한 방법대로 undo 가 수행되면, 서버에 있는 데이타 페이지에 로그 레코드 +10, +20, +30를 이용하여 세 번의 undo 가 발생하기 때문에 완전 철회 후의 페이지 내의 레코드의 값은 70이 되어 실제로 원래 값인 120과 다르게 된다. 이것은 서버에 있는 데이타 페이지는 +10에 의한 갱신 효과만 있는 상태에서 +10, +20과 +30에 해당하는 undo 를 수행하였기 때문에 발생하는 일관성 파괴현상이다. 그러므로 ESM-CS 에서는 트랜잭션 철회시 undo에 사용될 로그 레코드의 LRC 와 실제 데이타 페이지의 pageLRC 를 비교하여 pageLRC 가 로그 레코드의 LRC 보다 크거나 같을 때만 undo 하는 조건부 undo 를 수행한다.

ESM-CS 에서는 비록 pageLRC 가 로그 레코드의 LRC 보다 작아서 undo 가 수행되지 않더라도, 그것에 해당하는 상쇄 로그 레코드는 로그에 기록하되, 이 상쇄 로그 레코드의 LRC 를 데이타 페이지의 pageLRC 에 기록시키지 않는다. 이것은 조건부 undo 시 발생될 수 있는 여러가지 문제점을 해결하기 위해서이다[5].

4.2.3 Redo 단계

Redo 작동을 위해서 분석 단계에서 redo 가 시작되는 지점을 계산하여야 한다. redo 가 시작되는 지점을 계산하기 위해서는 각 데이타 페이지에 대해 이것들을 처음으로 dirty 시킨 연산의 로그 레코드의 LSN 를 알아야 하기 때문에, ESM-CS 는 클라이언트가 서버로 부터 받은 데이타 페이지에 첫번째 갱신이 발생하였을 때, 생성되는 로그 레코드의 LRC[8] 를 따로 기록하였다가 갱신된 페이지가 다시 서버로 보내질 때, 이 recoveryLSN 근사치를 같이 보내, 서버가 이것을 해당 페이지의 RecLSN 로 사용하게끔 한다.

[8] 이것을 recoveryLSN 근사치(*approximate recoveryLSN*)라 부름.

ESM-CS 의 서버는 dirty 된 페이지의 서버 도착 사실로 부터 해당 페이지가 dirty 되었음을 DPT 에 기록하기 때문에, 이 방법은 다음과 같은 두가지 문제점을 발생시킨다.

첫째는 로그 레코드의 서버 도착 시각와 해당 dirty 데이타 페이지의 서버 도착 시각 사이에 검사점 설치가 이루어지면, 검사점 로그 레코드의 DPT 에 해당 페이지의 dirty 사실이 기록되지 않아, 재 가동 고장회복시 완료된 트랜잭션에 필요한 redo 가 수행되지 않을 수 있다. ESM-CS 에서는 이러한 문제점을 해결하기 위해 완료 dirty 페이지 목록(*Commit Dirty Page List*)을 도입하여, 완료되는 트랜잭션의 완료 dirty 페이지 목록을 로깅하게 하여, 분석 단계 중에 고장회복 관리자가 완료 dirty 페이지 목록으로부터 필요한 redo 정보를 얻을 수 있게 하였다.

둘째는 첫번째 문제점에서와 같은 상황에서 만일 트랜잭션이 완료되기 전에 시스템 고장이 발생하면, 완료되지 못한 트랜잭션을 위한 완료 dirty 페이지 목록이 기록되지 않으므로, 첫번째 문제점의 해결책을 사용하여도 계속적으로 redo 되지 않는 로그 레코드가 존재하게 된다. 이러한 문제점을 발생시키는 트랜잭션은 모두 시스템 고장 발생시 완료되지 못한 트랜잭션이므로, ESM-CS 에서는 트랜잭션 테이블의 각 항목에 *StartLSN* 필드를 두어, 발생되는 문제점을 해결도록 하였다. 지면 관계상 위 두가지 문제점에 대한 자세한 설명과, 각각의 해결책에 대한 자세한 설명은 하지 못하지만, 관심있는 독자는 [5]를 참조하기 바란다.

4.3 제안된 방법

본 논문에서는 클라이언트-서버 DBMS 환경하에서 [5]에서 제안한 방법과는 다른 고장회복 기법을 제안한다. 근본적으로 가정하는 클라이언트-서버 DBMS 의 구조는 ESM-CS 의 그것과 같다. 그리고 제안하는 방법 역시 시스템 설계의 복잡성을 탈피하고자 클라이언트가 서버로 데이타 페이지를 전송할 때 WAL 규약을 따른다고 가정한다. 이 가정은 현재처럼 클라이언트에서 대용량의 메모리 버퍼를 사용하는 시스템에서는 큰 부담을 유발하지 않을 뿐더러, WAL 규약을 사용하지 않을 경우에는 서버에서 이를 지원하기 위한 여러가지 bookkeeping 부담을 고려한다면, 이러한 가정은 적절하다고 볼 수 있다. 그리고 ESM-CS 에서와 같이 클라이언트에서의 WAL 규약을 위해 LRC 개념을 사용한다. 그러나 본 논문에서 제안한 방법은 LRC 를 단순히 클라이언트에서의 WAL 규약 구현만을 위해서 사용하지만, ESM-CS 에서는 이것을 서버에서도 사용된다. 그러므로 본고에서의 LRC 는 클라이언트에서만 사용되기 때문에, ESM-CS 에서의 방법과는 달리 서버와 클라이언트에 의해 독립적으로 관리된다[9].

4.3.1 부분 철회 및 전체 철회

클라이언트가 자신이 수행하던 트랜잭션을 철회하고자 할 때는, 먼저 그때까지 트랜잭션에 의해 생성된 모든 로그 레코드를 서버로 전송하고, 이때 철회 요구 메세지를 함께 전송한다. 서버는 클라이언트로부터 트랜잭션 철회 요구 메세지를 받으면, 4.2.2 절에서 설명한 조건부 철회를 실

[9] ESM-CS 에서는 옳바른 LRC 를 생성하기 위해 서버가 클라이언트로 데이타를 전송하거나, 클라이언트에서 서버로 데이타를 전송할 때, LRC (혹은 LSN) 에 대한 정보를 함께 전송하여, 일종의 LRC(LSN) 동기화를 수행한다. 그리고 본 논문에서 제안한 클라이언트가 LRC 를 생성하는 방법은 4.3.2 절에서 설명된다.

시한다. 이 때, 클라이언트로부터 받은 철회 요청이 부분 철회라면, 철회가 기존에 설치된 안전점 로그 레코드까지 수행되고, 요청이 전체 철회이면, undo 에 사용되는 로그 레코드의 PrevLSN (혹은 UndoNxtLSN)이 0일 때까지 수행된다. 트랜잭션 철회가 완료되면, 서버는 이 사실을 클라이언트에게 알린다. 서버로부터 트랜잭션 철회 완료 메세지를 받으면, 철회된 트랜잭션이 갱신시킨 데이타 페이지 중 클라이언트 버퍼에 있는 페이지들을 버퍼 풀에서 제거한다.

이 방법은 두가지 단점을 갖는다. 첫째, 시스템의 병렬성을 충분히 사용하지 못한다. 만일 클라이언트가 자신이 갖고 있는 로그 레코드를 이용하여 자체적으로 트랜잭션 철회를 수행한다면, 트랜잭션에 대한 철회가 두 노드(트랜잭션의 갱신효과가 이미 서버에 반영된 경우에는 서버에서, 그렇지 않는 경우에는 클라이언트 자체적으로)에서 동시에 실시되므로 보다 높은 병렬성을 얻을 수 있다. 그러나 이러한 병렬성을 지원하기 위해서 클라이언트가 고장회복에 대한 코드를 갖고 있어야 하는 부담과 클라이언트와 서버와의 동기화에 대한 부담[10]을 고려해 볼 때, 여기서 채택한 방법이 갖고 있는 단점은 그리 심각하다고 볼 수 없다고 생각된다.

둘째, 클라이언트가 서버로부터 철회 완료 메세지를 받으면, 그 트랜잭션에 의해 변경된 버퍼 슬롯을 모두 무효화(invalidate)시키기 때문에 데이타 캐슁의 효과가 떨어지게 된다. 만일 클라이언트에서 자주 사용되는 데이타(예를들면 시스템 카다로그 또는 인덱스)가 있는 경우에는 클라이언트에서 캐쉬 적중률이 심하게 저하되어 클라이언트와 서버 사이의 메세지 양이 급격히 증가하게 될 수 있다. 그러나 클라이언트가 자체적으로 트랜잭션을 철회하는 루틴을 갖고 있지 않는 한, 이러한 문제점을 해결할 수 있는 방법은 없다. 다만, 클라이언트에서 자주 사용되는 데이타 페이지에 대해서만 여러개의 버젼을 두는 방법을 사용하여 클라이언트와 서버 사이의 메세지 양을 줄일 수는 있다.

트랜잭션 철회는 서버가 시작시킬 수도 있다. 예를들어 교착상태는 서버에 의해서 감지되기 때문에 트랜잭션이 교착상태(deadlock)에 빠졌을 때는 서버가 희생자를 골라 해당 트랜잭션을 철회시켜야 한다. 서버가 시작시키는 트랜잭션 철회는 먼저 해당 트랜잭션이 수행되고 있는 클라이언트에게 트랜잭션을 철회 요청을 보내므로써 해결된다. 철회 요청을 받은 클라이언트는 앞서 설명한 클라이언트 시발 트랜잭션 철회(*client-initiated transaction rollback*) 방법에 의해서 undo 가 수행된다.

여기서 트랜잭션 철회를 실제로 처리하는 서버가 직접 트랜잭션 철회를 실시하지 않고, 해당 클라이언트에게 철회를 요구하는 이유는 만일 서버가 독자적으로 트래잭션 철회시키고 이 사실은 클라이언트에게 알렸을 때, 클라이언트는 그 트랜잭션이 갱신시킨 페이지 뿐만이 아니라, 그 때까지 로그 버퍼에 있는 해당 로그 레코드도 제거해야 되는데, 한 로그 버퍼 페이지에 다른 트랜잭션의 로그 레코드가 있을 수 있으므로, 해당 로그 레코드만을 찾아 제거시키는 부담을 없애기 위해서이다.

[10] 예를들면 동기화를 위한 클라이언트의 복잡도 증가와 클라이언트와 서버사이에 전달되는 메세지 양의 증가를 들 수 있다.

4.3.2 갱신 작동 및 페이지 전송 작동

본 논문에서는 WAL 규약을 구현하기 위해, 모든 클라이언트가 각자 관리하는 "*ClientLRC*" 라는 계수기(counter)가 있다고 가정하고, 클라이언트의 버퍼 제어 블럭에 "*BufferLRC*" 라는 필드가 있다고 가정한다[11]. 다음은 클라이언트에서 갱신 작동이 발생할 때 어떠한 작업이 수행하는지를 보여주고, 클라이언트와 서버 사이에 데이타 전송이 일어날 때 어떠한 작업이 수행되는 지를 보여준다.

클라이언트에서 한 레코드에 대한 갱신 연산을 수행할 때는 다음과 같은 작업이 수행된다.

1. 해당 페이지가 클라이언트 버퍼에 있는가 조사하여, 없으면 서버에 요청한다. 이때, 요청한 페이지가 도착하면 전송받은 데이타를 빈 버퍼 슬롯을 할당받아, 그곳에 저장하고 그 버퍼 슬롯 제어 블럭의 BufferLRC 를 현재 ClientLRC 값으로 초기화시킨다.

2. 해당 레코드에 갱신 연산을 수행한다.

3. 갱신에 해당하는 로그 레코드를 생성하여, 이 로그 레코드의 LRC 필드와 갱신된 레코드가 포함된 페이지가 위치한 버퍼 슬롯의 제어 블럭의 BufferLRC 에 현재 ClientLRC 값을 기록한다.

4. 사용하고 있는 로그 페이지에 생성된 로그 레코드를 (공간부족 등의 이유로) 삽입시킬 수 없으면, 새로운 로그 페이지를 할당받아, 초기화(log page formatting)시키고 해당 버퍼 슬롯의 BufferLRC 에 로그 레코드의 LRC 를 기록한다.

5. ClientLRC 계수값을 하나 증가시킨다.

클라이언트의 데이타 버퍼 페이지가 서버로 전송될 때는 먼저 로그 버퍼 슬롯 제어 블럭들의 BufferLRC 값이 전송시킬 데이타 버퍼 슬롯의 제어 블럭의 BufferLRC 보다 같거나 작은 모든 로그 버퍼 페이지를 데이타 페이지와 함께 전송하는 방법으로 클라이언트에서의 WAL 규약을 구현한다.

서버가 클라이언트로부터 특정 페이지의 요청을 받으면 해당 페이지의 잠금장치를 잠근 후, 페이지를 클라이언트로 전송한다. 서버가 클라이언트로 부터 로그 레코드를 받을 때는 다음과 같은 동작을 취한다.

1. 로그 레코드를 로그 화일에 저장하고, 로그 레코드에 LSN 을 부여한다.

2. 로그 레코드의 TransID 를 이용하여 트랜잭션 테이블을 검색한다.

   - 트랜잭션 테이블에 해당 항목이 없으면, TransID를 이용하여 트랜잭션 테이블에 등록하되, 이때 LastLSN 과 UndoNxtLSN 에 로그 레코드의 LSN 을 기록하고, 로그 레코드의 PrevLSN 을 0로 기록한다.

[11] BufferLRC 는 데이타 페이지 버퍼 제어 블록과 로그 페이지 버퍼 제어 블록 모두 있다고 가정한다.

- 해당 항목이 검색되면, 로그 레코드의 PrevLSN 에 검색된 항목의 LastLSN 을 기록하고, 항목의 LastLSN 과 UndoNxtLSN 에 로그 레코드의 LSN 을 기록한다.

3. 로그 레코드의 PageID를 이용하여 DPT를 검색한다.

   - DPT에 해당 항목이 없으면, PageID를 이용하여 DPT에 등록하고, 이때의 RecLSN 과 LastLSN 에 로그 레코드의 LSN 을 기록한다.
   - 해당 항목이 존재하면, 검색된 항목의 LastLSN 필드를 로그 레코드의 LSN 으로 기록한다.

여기서 주의깊게 볼 점은 본 논문에서 제안하는 방법에서는 서버가 클라이언트로부터의 로그 레코드의 도착되면 해당 데이타 페이지가 dirty 된 것으로 간주한다(즉, DPT 에 등록한다). 이것은 실제로 dirty 된 페이지가 서버에 도착했을 때 DPT 에 등록하는 ESM-CS 에서 방법과는 차이가 있다. 이러한 차이로 부터 발생되는 결과는 4.4 절에 기술되어 있다.

참고로 위 동작들은 비록 한 로그 레코드에 대한 동작들로 표현되어 있어, 클라이언트로부터 로그 페이지를 전송받으면, 로그 페이지에 있는 각 로그 레코드에 대해 위의 작동을 순서대로 취하면 된다. 물론 이러한 경우에는 약간의 불필요한 작업[12]이 있을 수 있지만, 본 논문에서는 독자들의 이해의 편의를 위해 알고리즘의 논리적인 기술에 중점을 두었다.

서버가 클라이언트로부터 데이타 페이지를 전송 받으면, 먼저 로그 페이지가 함께 도착하였나 조사하여 함께 도착한 로그 페이지가 있으면, 위와같은 방법으로 먼저 로그 레코드를 기록하고, 데이타 페이지를 저장하고, 페이지의 식별자를 이용하여 DPT를 검색하여, 해당 항목의 LastLSN 을 페이지의 pageLSN[13] 에 기록한다.

4.3.3 재 가동 고장회복

클라이언트-서버 DBMS 에서는 클라이언트와 서버가 서로 독립적으로 작동하기 때문에, 시스템 고장이 전체 시스템을 멈추게 하지 않는다. 그러므로 시스템 고장시 고장회복이 복잡해질 수 있다[1]. [5]에서는 이러한 재 가동 고장회복 작동(특히 고장이 발생하였을 때, 상대되는 시스템이 이를 대처하는 작동)에 대한 언급이 명확히 나와있지 않다.

본 논문에서는 고장회복 기법의 단순화를 위해 다소의 가용성(availability)을 제한하는 방법을 제안한다. 그리고 클라이언트와 서버는 각각 상대방의 시스템 고장 여부를 판단할 수 있다고 가정한다.

먼저 클라이언트가 서버에서의 시스템 고장을 감지하면, 수행 중이던 모든 트랜잭션을 철회시킨다. 이 때 트랜잭션을 철회시키는 방법은 3.2.2 절 또는 4.2.2 절에서 설명한 방법을 사용하

[12] 예를들면, 같은 페이지를 갱신하여 발생된 로그 레코드가 한 로그 페이지에 여러개가 있을 경우에는 그 중 맨 마지막 로그 레코드의 LSN 만을 DPT 의 LastLSN 에 기록한다거나, 또는 (TransID에 대한 래치를 계속적으로 갖고 있으면,) 불필요하게 여러번 TransID 를 검색할 필요가 없다.

[13] 본 논문에서 제안한 방법은 LRC 가 서버에서 전혀 사용되지 않기 때문에 기존의 ARIES 와 같이 데이타 페이지는 pageLSN 필드를 갖는다.

지 않고, 단순히 모든 데이타 버퍼 페이지와 모든 로그 페이지를 제거시킴으로써 클라이언트에서의 트랜잭션 철회가 완료된다. 즉, 서버 시스템의 고장이 발생하면, 클라이언트에서도 시스템 고장이 발생한 것과 같은 효과를 발생시킨다. 이러한 방법은 클라이언트가 이미 캐싱한 데이타 페이지를 무효화하는 일이 발생되고, 불필요한 트랜잭션 철회가 일어날 수도 있지만, 서버의 재가동 고장회복을 알고리즘을 간단하게 할 수 있고, 서버가 정상작동 중에 기록해야 하는 정보의 양을 줄일 수 있다. 클라이언트는 서버 컴퓨터가 다시 가동될 때까지 대기하여, 다시 서버 접속이 가능하면, 다시 트랜잭션을 처음부터 수행시키는 방법을 사용할 수 있다. 서버가 고장에서 회복될 때는 이미 클라이언트에서 수행되던 모든 트랜잭션이 철회되었기 때문에 정상작동시 트랜잭션 철회에 사용된 조건부 철회가 필요없고, 기존의 무조건 undo 를 undo 단계에서 사용할 수 있다. 이것은 undo 전 단계인 redo 단계에서 로그 레코드를 이용하여 모든 갱신 효과를 다시 반영시키기 때문이다. 무조건 undo 는 undo 단계에 걸리는 시간을 조건부 undo 보다 효과적으로 줄일 수 있다.

서버가 한 클라이언트의 고장을 확인하면, 해당 클라이언트에서 수행시키고 있던 모든 트랜잭션들을 철회시킨다. 여기서 서버가 클라이언트의 트랜잭션을 철회시키는 방법은 앞서 4.3.1 절에서 설명한 바와같이 해당 클라이언트에게 트랜잭션 철회를 요청할 수 없기 때문에 현재까지의 정보만을 이용하여 해당 클라이언트에서 수행하던 트랜잭션을 전체 철회시킨다. 이때는 4.2.2 절에서 설명한 조건부 철회 기법을 사용하여야 한다. 클라이언트는 고장에서 회복되면, 다시 서버와의 접속을 시도한다.

4.4 다른 관련 연구와의 비교

이절에서는 본 논문에서 제안한 클라이언트-서버 DBMS 고장회복 기법과 기존의 관련된 연구 주로 ESM-CS 에서의 방법[5]과의 비교를 자세하게 설명한다.

먼저 본 논문에서 제안한 고장회복 방법과 ESM-CS 에서의 방법은 모두 ARIES 고장회복 기법을 클라이언트-서버 환경으로 확장시켰다는 점과 클라이언트에서의 WAL 규약을 구현하기 위해 물리적인 LSN 외에 새롭게 논리적인 로그 레코드 식별자 LRC를 도입하였다는 점에서 동일하다. 그리고 로그 레코드와 해당되는 데이타 페이지의 서버로의 도착시각의 차이로 발생가능한 비일관성을 극복하기 위해 조건부 undo 를 사용하는 점도 같다.

그러나 다음은 본 논문에서 제안한 방법과 [5]의 방법과의 차이를 보여준다.

- 본 논문에서의 방법의 LRC 가 단순히 클라이언트에서의 WAL 규약 만을 구현하기 위해 사용되는 반면, [5]에서는 서버에서도 LSN 가 로그 레코드를 직접 접근하는데 사용되는 경우를 제외하고는 LRC를 사용한다.

- 본 논문에서는 로그 레코드가 서버에 도착하였을 때를 해당 페이지가 dirty 된 것으로 간주하는 반면, [5]에서는 실제로 dirty 된 페이지가 서버로 도착되었을 때 비로소 해당 페이지가 dirty 된 것으로 간주한다.

- 본 논문에서 제안하는 방법은 클라이언트 독자적으로 LRC 를 생성하지만, [5]에서는 서버

에서 클라이언트로 데이타를 전송할 때, end-of-LSN 을 함께 전송하는 방법으로 LRC 를 생성한다.

- 클라이언트에서의 WAL 규약을 구현할때, 본 논문에서는 전송하려는 데이타 페이지의 BufferLRC 와 로그 페이지의 BufferLRC 를 비교하는 방법으로 구현하지만, [5]에서는 전송하려는 데이타 페이지의 pageLRC 와 로그 페이지 내의 각 로그 레코드의 LRC 를 비교하는 방법으로 구현한다.

이로 인하여 두가지 방법에는 다소의 차이가 있다. 본 논문에서 제안한 방법은 [5]에서 제안한 방법과 비교할 때, 다음과 같은 장점을 갖는다.

1. LRC가 오직 클라이언트에서만 사용되기 때문에, [5] 에서의 방법에서와 같이 서버가 페이지를 클라이언트로 전송할 때, 현재 마지막 로그의 LSN 을 같이 전송할 필요가 없다. 즉, 클라이언트가 독자적으로 LRC를 관리할 수 있기 때문에, 보다 높은 독립성이 보장된다.

2. 클라이언트에서 LRC를 이용하여 WAL 규약을 관리하는 방법을 구현할 때, [5]에서 제안한 방법은 전송되어야 할 페이지의 pageLRC 와 로그 페이지 내의 각 로그 레코드의 LRC 를 비교해야 하는 과정이 필요하기 때문에 많은 오버헤드가 따른다. 그러나 본 논문에서 제안하는 방법은 전송해야 할 페이지가 있는 슬롯의 제어 블럭에 있는 BufferLRC와 로그 버퍼 슬롯의 제어 블럭에 있는 BufferLRC와 비교하기 때문에, 매우 효율적으로 WAL 규약을 구현할 수 있다.

3. [5]에서 제안한 방법을 사용하기 위해서는 클라이언트에서 생성되는 모든 로그 레코드는 LRC 필드를 필요로 한다. 이것은 로그의 낭비를 초래한다. 실제로 [5]의 성능평가 결론에 의하면, 로그 레코드의 크기가 로깅 오버헤드에 많은 영향을 주는 것을 말하기 때문에, 이것은 단순한 자원의 낭비일 뿐만 아니라, 추가의 로깅 오버헤드를 발생시킨다.

4. [5]에서 제안한 방법에서는 RecLSN 근사치를 사용하는 반면, 본 논문에서는 정확한 RecLSN 을 사용하기 때문에 불필요한 로그 화일 스캔을 유발하지 않는다.

5. 본 논문의 방법은 로그 레코드 도착이 페이지 dirty 여부를 결정하기 때문에, 4.2.3 절에서 언급한 두가지 문제점이 발생하지 않으므로, 종료 dirty 페이지 목록이나 StartLSN이 불필요하고, 분석 단계도 한결 단순하다.

위에서 언급한 바와같이 본 논문에서는 로그 레코드의 도착을 해당 페이지에 대한 dirty 사실로 간주하기 때문에, 매 로그 레코드가 도착할 때마다 DPT 를 검색해야 하는 오버헤드가 있다. 이러한 오버헤든 서버가 클라이언트로부터 로그 레코드를 받을 취하는 동작을 페이지 단위로 바꾸면 어느정도 오버헤드를 감소시킬 수 있다(주석 12 참조).

5 결론 및 연구 방향

강력하고 저렴한 워크스테이션의 등장과 고속의 네트워크 통신 기술이 가능해 짐에 따라, 많은

관심을 받기 시작한 클라이언트-서버 DBMS 는 현재 여러가지 측면에서 많은 연구가 되어지고 있다. 그러나 클라이언트-서버 DBMS 의 고장회복에 관한 연구는 아직까지는 깊이 있게 연구되지 않고 있는 실정이다.

본 논문은 현재 고장회복 기법으로 많은 관심을 받고 있는 ARIES 고장회복 알고리즘을 클라이언트-서버 구조 DBMS 로 확장시켰을 때, 발생 가능한 문제점에 대한 새로운 해결방법을 제시하였다. 그리고 본 논문에서 제안한 기법과 기존의 기법과의 비교를 통해, 본 논문에서 제안한 방법이 기존의 방법보다 우수하다는 것을 보였다.

앞으로는 연구될 것으로는 첫째, 클라이언트-서버 DBMS 에서 로깅 기능이 차지하는 오버헤드에 대한 연구가 필요하다. 이것을 위해서는 시뮬레이션 기법을 이용하여, 클라이언트-서버 DBMS 가 로깅을 할 때와 하지 않았을 때의 성능 비교를 통해 가능하다. 또한 ESM-CS 에서 사용한 방법과의 성능 평가도 이루어질 것이다.

본 논문에서 제안한 고장회복 기법은 아직 구현되지 않았으나, 우리가 있는 연구실에서 앞으로 개발할 클라이언트-서버 DBMS 에 사용될 것이다. 이러한 클라이언트-서버 DBMS 가 개발되면, 벤치마크 테스트를 통해, 좀더 정확한 성능평가 결과를 알 수 있을 것이다.

참고 문헌

[1] P. A. Berstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and in Database Systems*. Addison-Wesley Publishing Company, 1987.

[2] R. A. Crus. "Data recovery in IBM Database 2". *IBM Syst. J.*, 23(2):178–188, 1984.

[3] D. DeWitt, D. Maier, P. Futtersack, and F. Velez. "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems". In *Proc. of the Conf. on VLDB*, pages 107–121, Brisbane, Australia, Aug. 1990.

[4] EXODUS Project Group. "EXODUS Storage Manager Architectural Overview". Technical report, Computer Science Dept., University of Wisconsin-Madison, Nov. 1991.

[5] M. J. Franklin, M. J. Zwilling, C. K. Tan, M. J. Carey, and D. J. DeWitt. "Crash Recovery in Client-Server EXODUS". In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 165–174, San Diego, June 1992. Also available as Technical Report #1081, Comp. Sci. Dept. University of Wisconsin-Madison, Mar. 1992.

[6] J. Gray et al. "The Recovery Manager of the System R Database Manager". *ACM Comput. Surv.*, 13(2):223–242, June 1981.

[7] T. Haerder and A. Reuter. "Principles of Transaction-Oriented Database Recovery". *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[8] M. F. Hornick and S. B. Zdonik. "A Shared Segmented Memory System for an Object-Oriented Database". *ACM Trans. Office Inf. Syst.*, 5(1):70–95, Jan. 1987.

[9] H. F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, Inc, second edition, 1991.

[10] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. "The ObjectStore Database System". *Commun. ACM*, 34(10), Oct. 1991.

[11] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communication of the ACM*, 21(7):558–565, July 1978.

[12] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.