ARIES/RL: 장기간 트랜잭션을 지원하기 위한 ARIES 확장
(ARIES/RL : An Extension of ARIES with Re-Logging to Support Long-Duration Transactions)

요 약

본 논문에서는 장기간 트랜잭션이 존재할 경우에, 제한된 로그 공간을 효율적으로 사용하기 위해 재로깅이라는 기법을 ARIES에 확장한 ARIES/RL 알고리즘을 제시한다. ARIES/RL은 로그 공간이 현재 수행되는 트랜잭션에 충분하지 않은 경우 트랜잭션 취소나 재시동 고장회복에 사용되는 로그레코드를 로그의 끝쪽으로 옮기는 기법이다. 이 기법은 기존의 ARIES가 가지는 장점을 그대로 유지하면서, 장기간 트랜잭션을 수행시키는 경우 로그를 더 효율적으로 사용하게 된다. 논문에서는 ARIES/RL의 성능 평가의 결과를 제시한다.

주제어 : 데이타베이스, ARIES/RL, 재로깅, 고장회복, 성능평가

Abstract

We propose the ARIES/RL which extends the ARIES with 're-logging' technique to manage the limited online log space efficiently even though long-duration transactions exist.

Re-logging is a technique that log records used in transaction rollback and restart recovery are copied forward in the log, whenever the online log is not sufficient for keeping logs of on-going transactions. It does not hurt the advantages of ARIES. Moreover, it handles log space efficiently in executing long-duration transactions.

We also present the evaluation result of ARIES/RL and show that ARIES/RL handles online log efficiently.

Keyword: database, ARIES/RL, re-logging, recovery, evaluation

1 서론

로그는 트랜잭션 고장, 시스템 고장, 장치 고장 발생시 고장회복을 위해 사용된다. 트랜잭션이 데이타베이스를 수정하거나, 데이타의 일관성을 유지하기 위한 중요한 일을 할 때 DBMS는 로그 레코드를 기록한다. 여러 고장상황에서 데이타베이스를 일관성 있는 상태로 만들기 위해 로그가 사용된다. 로그가 저장되는 로그 공간은 대부분의 DBMS에서 가장 큰 저장 공간 중의 하나이며, 데이타베이스 시스템에서 성능상 병목현상을 야기하고 있다[11,18].

새로 등장되고 있는 사무자동화(OA), CAD/CAM등에서 쓰이는 여러 응용 프로그램들은 장기간 트랜잭션(long duration transaction:LDT)을 필요로 하고 있다. 기존의 DBMS시스템에서는 LDT에

대한 지원이 미비하여 이 부분에 대한 많은 연구들이 되고 있다[2, 6, 9]. 이 논문에서는 이런 장기간 트랜잭션이 있는 경우 온라인 로그공간의 효율적인 운영에 초점을 맞추었다.

우리는 ARIES 알고리즘에 재로깅 기법을 이용해 확장한 ARIES/RL(ARIES with **Re**-**L**ogging) 알고리즘을 제시한다. ARIES/RL은 사용자가 수행한 트랜잭션 중 LDT가 있는 경우에 온라인 로그 공간을 효율적으로 관리한다. ARIES는 트랜잭션 관리 시스템에서 간단하고 효율적이라고 알려져있는 고장회복 알고리즘이다[20]. ARIES는 다른 고장회복 알고리즘에 비해 많은 장점을 가지고 있어 많은 상업용 DBMS나 연구용 DBMS에서 구현되고 있다[4, 5, 12].

논문의 나머지 부분은 다음과 같이 구성되어 있다. 2 장에서는 논문의 동기와 기본 개념에 대해서 설명한다. 3장에서는 ARIES/RL의 자료구조와 알고리즘에 대해 설명한다. 4장에서는 ARIES/RL의 성능 평가 결과에 대해서 제시한다. 5장에서는 LDT가 있는 경우 로그 공간 활용에 대한 관련연구에 대해 비교 설명하고 6장에서 결론에 대해 정리한다.

2 기본 개념

2.1 용어 설명

먼저 본 논문에서 사용되는 몇가지 용어에 대해서 설명하겠다. ARIES에 관련된 용어들은 [20] 논문의 관례를 따랐다. 우리는 독자가 ARIES에 대해서 익숙한 것으로 가정한다.

트랜잭션의 *FirstLSN(T)*은 트랜잭션 T에 의해서 처음으로 만들어진 로그레코드의 LSN이다. *BeginChkptLSN*은 가장 최근의 성공한 검사점 연산에 의해 만들어진 '`BEGIN_CHKPT`' 로그 레코드의 LSN이다. *RedoLSN*은 버퍼 풀 변경 페이지 테이블에 있는 RecLSN들 중 제일 작은 LSN값이다. 재시동 고장회복시에 여기서부터 재수행 단계를 시작하게 된다. RedoLSN보다 적은 LSN값을 가지는 로그 레코드들은 재시동 고장회복시 재수행 단계에서는 접근되지 않는다. 그리고 이 값은 버퍼를 주기적으로 디스크에 반영함으로써 증가시킬 수 있다. RedoLSN과 '`BEGIN_CHKPT`'보다 적은 LSN값을 가진 로그레코드들은 트랜잭션 취소나 재시동 철회 단계에서만 접근된다. 트랜잭션 T의 FirstLSN(T)이 RedoLSN과 '`BEGIN_CHKPT`'보다 적은 트랜잭션에 대해, RedoLSN과 '`BEGIN_CHKPT`' 중 최소값과 T의 FirstLSN(T)과의 차이를 T의 *UndoOverhead(T)*로 정의한다. 데이타베이스에서 빠른 고장회복을 지원하기 위해서 수행중인 트랜잭션 T의 UndoOverhead(T)내에 있는 로그 레코드들은 오프라인 로그 공간으로 옮겨질 수 없다. *UndoLSN*은 현재 수행되는 모든 트랜잭션의 FirstLSN(T)들 중 가장 작은 LSN이다. *RecoveryLSN*은 RedoLSN, UndoLSN, 그리고 '`BEGIN_CHKPT`' 중 가장 작은 LSN 값이다. RecoveryLSN보다 적은 LSN 값을 가지고 있는 모든 로그 레코드는 트랜잭션 취소나 재시동

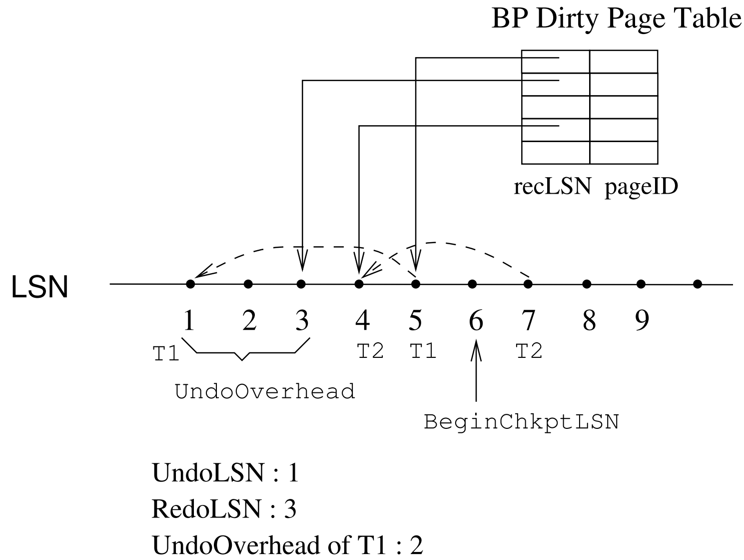


그림 1: 몇가지 용어 예제

고장회복시에 접근되지 않는다. 그러므로 온라인 로그 공간을 절약하기 위하여 RecoveryLSN보다 적은 LSN 값을 갖는 로그 레코드들은 오프라인 저장장치로 옮기거나 버려질 수 있다. RecoveryLSN은 System R의 **FIRE_WALL**[10]과 같은 개념이다. 이해를 돕기 위해 그림 1을 보자. 로그 레코드 3과 1은 각각 RedoLSN과 UndoLSN을 나타낸다. 트랜잭션 **T1**의 UndoOverhead(T1)는 LSN 1과 LSN 3 사이의 공간의 차를 나타낸다.

2.2 관찰 사항

대부분의 데이타베이스 시스템들은 트랜잭션 취소나 재시동 고장회복을 위해 사용되는 로그를 빠른 온라인 저장장치에 저장한다. 그리고 나머지 로그들은 테입과 같은 대량의 오프라인 저장장치에 저장한다. 즉, LSN이 RecoveryLSN보다 큰 모든 로그 레코드들은 온라인 로그 공간에 유지된다. 온라인 로그 공간을 줄이기 위해 DBMS는 주기적으로 검사점 연산을 수행하여 변경 페이지들은 디스크 공간에 저장해야 한다. 이러한 연산들은 통해서 **BeginChkpLSN**과 RedoLSN을 증가시킬수 있다[20]. 대부분 상대적으로 짧은 트랜잭션들이 수행되는 기존 데이타베이스 응용 프로그램에서는 이 **BeginChkpLSN**과 RedoLSN을 증가시킴으로서 RecoveryLSN을 증가시킬 수 있다.

그러나, 만약 장기간 트랜잭션이 존재하는 경우 UndoLSN이 RecoveryLSN의 증가를 막게 된다. 더우기 트랜잭션을 취소시키는 극단적인 방법을 사용하지 않고는 데이타베이스 시스템에서 UndoLSN을 증가시킬 방법이 없다. 한 트랜잭션의 로그레코드들이 다른 트랜잭션 T의 UndoOverhead(T)사이에 있다면 이미 완료된 트랜잭션들에 의해 생성된 로그레코드들조차 오프라인 저장장

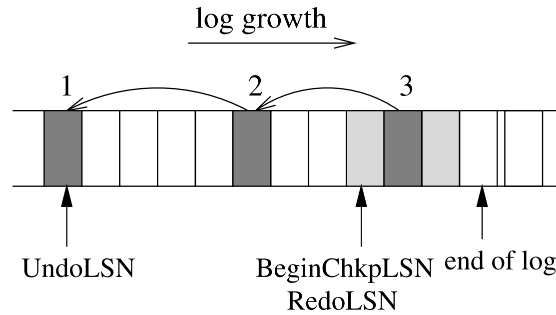


그림 2: 장기간 트랜잭션에 의해 발생한 로그 공간 부족 현상

치로 옮겨질 수 없다. System R에서는 옮겨질 로그 레코드의 시점을 나타내기 위해서 FIRE_WALL이라는 기준점을 사용했다. FIRE_WALL이후의 모든 로그 레코드들은 온라인 환형 버퍼 상에 존재해야만 한다.

그림 2를 보자. 음영으로 표시된 칸이 현재 수행중인 트랜잭션의 로그레코드들이다. 그리고 흰 칸이 이미 완료된 트랜잭션의 로그레코드들이다. 이 그림에서 로그 레코드 1이 여섯개의 흰 로그 레코드들을 오프라인 저장장치로 옮겨지는 것을 막기 때문에 온라인 로그에 여섯개의 사용되지 않은 로그 레코드들을 유지해야만 한다[1]. 로그 레코드 1을 기록한 트랜잭션이 끝나지 않을 경우, 점차로 온라인 고장회복에 사용되지 않은 로그레코드들의 갯수는 더 증가될 수 있다.

2.3 기본적 고찰 : 재로깅(ReLogging)

재로깅의 기본적인 아이디어는 간단하다. UndoLSN의 증가를 막는 로그레코드들을 로그의 맨 끝으로 옮겨 새로운 로그 레코드로 다시 기록하는 것이다[2]. 트랜잭션을 철회되는 경우 이 옮겨진 로그레코드가 기존의 로그레코드 대신 사용될 것이다. 그림 3을 보자. 로그레코드 1과 2를 대신하는 대체 로그레코드를 기록함으로써, 여섯개의 사용되지 않은 로그레코드들이 오프라인 저장장치로 옮겨질 수 있다.

ARIES에서 보상 로그 레코드(Compensate log record: CLR)가 가지는 장점들을 유지하기 위해서 그 효과가 아직 철회되지 않은 철회가능한(undoable) 로그레코드들만이 재로깅된다. 그림 4에서 보여지는 모든 로그레코드들은 동일한 장기간 트랜잭션 LT에 의해 기록되었다고 가정하자. LT는 데이타베이스를 네번 수정하여서, 철회가능한 로그레코드 1, 2, 3, 4를 기록하였다. 그리고 부분철회를 수행해 로그레코드 4와 3에 해당하는 수행한 연산을 철회했다. 그 후 계속 진행해 로그 레코드 5와 6을 기록하였다. 검사점 연산 수행 중에 UndoLSN을 증가시키기 위해 트랜잭션 T의 로그 레코

[1] 설명을 위해서 우리는 매체 고장회복(media recovery)는 고려하지 않았다.

[2] 이 옮겨진 로그레코드들을 대체 로그레코드(alternative log record)라고 부르겠다.

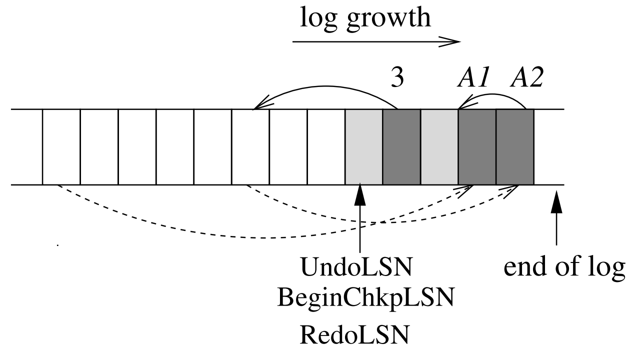


그림 3: 재로깅 후의 온라인 로그 상황

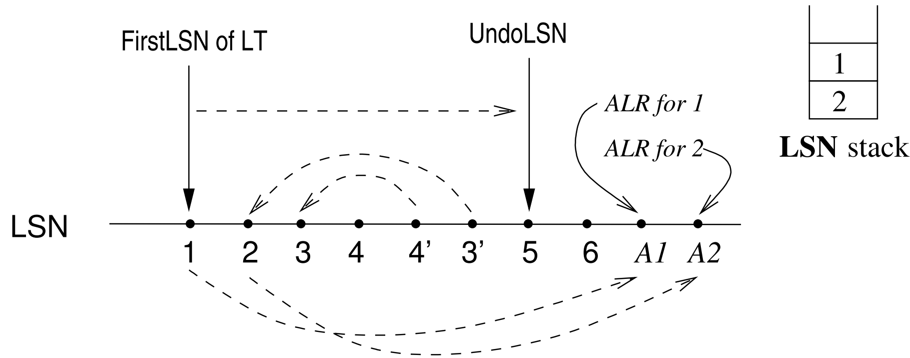


그림 4: 로그 분석후 재로깅

드 중 1, 2, 3, 4을 재로깅하기로 결정했다고 하자.[3]

그 효과가 아직 철회되지 않은 모든 철회 가능한 로그레코드들을 찾기 위해 시스템은 트랜잭션 LT에 의해서 생성된 로그레코드들을 검색한다. 이 과정은 트랜잭션 취소가 일어날 때 로그레코드를 검사하는 과정과 같다. 시스템은 트랜잭션 테이블의 UndoNxtLSN(그림에서 LSN 6)부터 로그레코드를 검사한다. 재수행용(redo-only) 로그레코드들과 이미 취소된 로그레코드들은 재로깅되지 않는다. 시스템이 옮겨져야 할 취소 가능한 로그레코드들을 발견할 때마다, 그 로그레코드들의 LSN을 올바른 재로깅을 보장하기 위해서 *LSN 스택*에 저장한다. 로그레코드에 대한 검사가 끝난 후, 시스템은 LSN 스택에 저장되어 있는 로그레코드들을 후입선출 순서(LIFO 순서)에 따라 재로깅을 수행한다. 취소 가능한 로그레코드들만이 대체 로그 레코드를 생성하는데 이용된다.

자세한 알고리즘은 다음 장에서 설명된다. 알고리즘을 구현하기 위해 필요한 자료 구조에 대해서 먼저 설명하겠다.

[3] RecoveryLSN을 결정하는 요인 중 RedoLSN, 'BEGIN_CHKPT'로그 레코드 값들은 주기적으로 버퍼를 디스크에 반영시키거나 검사점 연산을 자주 수행함으로써 증가시킬 수 있다.

3 ARIES/RL

3.1 ARIES/RL의 데이타 구조

이 장에서는 ARIES/RL을 자세히 설명하기 위해 필요한 주요 자료구조에 대해 설명하겠다.

*트랜잭션 테이블*은 정상 수행이나 재시동 고장회복 시 수행중인 트랜잭션의 정보를 기록한다. 이 트랜잭션 테이블의 엔트리 항목에 *firstALR*과 *lastALR*을 추가하였다. 재로깅된 트랜잭션 T의 FirstALR(T)은 T에 대한 재로깅 결과로 생성된 첫 로그 레코드의 LSN이다. 그리고 첫 대체 로그레코드가 다시 재로깅된다면, FirstALR(T)은 옮겨진 것을 가르킨다. LastALR(T)은 트랜잭션 T에서 가장 나중에 쓰여진 대체 로그레코드의 LSN이다. 이 값은 트랜잭션 철회과정이 재수행 고장회복시 재로깅된 로그레코드가 참조될 때 사용된다.

동일한 트랜잭션의 모든 대체 로그레코드들은 정상 로그레코드들처럼 PrevLSN을 통해서 연결되어 있다. 모든 대체 로그레코드들은 재로깅될 로그레코드의 LSN이 저장되어 있는 *originLSN* 필드를 가지고 있다. 대체 로그레코드의 *Type* 필드는 'ALTERNATIVE' 값을 가지고 있다.

데이타베이스의 모든 페이지에 그 페이지에 가장 최근에 갱신한 로그 레코드의 LSN을 나타내는 page_LSN 값을 헤더에 가지고 있다. 그리고 버퍼 풀 변경 페이지 테이블은 *PageID*와 *RecLSN* 값을 가지고 있다. 기존 ARIES 알고리즘에서 사용되는 자료구조와 동일하다.

3.2 정상 동작

3.2.1 검사점 연산

본 논문에서는 재로깅 알고리즘의 복잡도를 줄이기 위해, 재로깅은 검사점 설치 중에 일어난다. 검사점 연산은 재시동 고장회복시에 해야 할 일을 줄이기 위해 주기적으로 수행된다. 검사점 연산은 먼저 'BEGIN_CHKPT' 로그 레코드를 기록함으로써 시작된다. 검사점 설치 방법에 따라 검사점 연산 중에 오랫기간동안 디스크에 반영되지 않았던 페이지들이 디스크에 반영되기도한다. 그러나 모든 변경 페이지들을 디스크에 반영하지는 않는다. 수행중인 트랜잭션에 대한 정보와 버퍼 풀 변경 페이지 테이블이 수집된 후, 재로깅을 여부를 결정하여 수행한다. 기존 ARIES의 검사점 연산 수행 중, 버퍼 풀 변경 페이지에 대한 정보가 수집된 이후, 각 트랜잭션의 UndoOverhead(T)를 계산하여 재로깅을 하게 된다.

그 후에, 'END_CHKPT' 로그 레코드를 로그에 기록한다. 정상 트랜잭션 테이블, 버퍼 풀 변경 페이지 테이블, 그리고 화일 연관 정보(file mapping information)들의 내용들이 'END_CHKPT' 로그 레코드에 기록된다. 재로깅에 대한 모든 정보들은 트랜잭션 테이블에 포함되어 있다. 'END_CHKPT' 로

그 레코드가 로그에 기록되고 나면, 'END_CHKPT'로그레코드의 LSN이 미리 지정된 *매스터 로그 레코드(master log record)* 에 저장된다.

재로깅 알고리즘의 유사 코드가 그림 5에 기술되었다. 재로깅 루틴은 검사점 연산 수행중에 시행된다. 이 루틴에 대한 입력은 현재 수행중인 트랜잭션의 상태를 유지하는 트랜잭션 테이블과 UndoLSN의 옮겨질 위치를 지정하는 truncLSN 값이다. 온라인 로그 상에 남아 있어야 되는 로그의 위치를 표시하는 RecoveryLSN은 RedoLSN[4] , UndoLSN[5], 'BEGIN_CHKPT'[6] 값들의 최소값으로 결정이 된다. 이 값들 중 RedoLSN과 UndoLSN 값은 버퍼를 flush하거나 검사점 연산을 실시하는 작업을 통해서 시스템에서 임의로 증가시킬 수 있다. 그러므로 UndoLSN 값만 조절하면 RecoveryLSN 값을 조정할 수 있다. 만약 각 트랜잭션 T의 UndoOverhead(T)가 지정된 한계값보다 클 경우[7], 이 트랜잭션은 LDT로 판단하여 이 트랜잭션의 일부 로그 레코드들을 재로깅한다.

재로깅 루틴은 먼저 UndoNxtLSN에서부터 선택된 트랜잭션의 첫번째 로그 레코드까지 로그를 역방향으로 검색하면서 재로깅될 로그레코드들에 대해 분석한다(유사 코드의 2 ∼ 14문). 4 행은 분석할 로그 레코드가 이미 재로깅되어 옮겨졌을 경우 이 재로깅된 로그레코드를 참조하기 위해 필요하다. 만약 읽혀진[8] 로그레코드의 타입이 'UPDATE'이고 그 LSN이 truncLSN보다 적은 경우이거나, 로그 레코드의 타입이 'ALTERNATIVE'인 경우에 로그레코드의 LSN은 LSN 스택에 저장된다. 'ALTERNATIVE' 로그레코드는 그 LSN이 tructLSN보다 큰 경우에도 선택이 된다. 'ALTERNATIVE' 로그 레코드들을 전부 재로깅함으로써 재로깅되는 로그 양은 많아질 수 있으나, 재로깅된 로그 레코드가 다시 재로깅되는 상황에 대해서 특별하게 신경 쓸 필요가 없어지게 된다. 그림 4에서 로그레코드 2와 1의 LSN이 LSN 스택에 저장된다. 다음으로 LSN 스택에 있는 로그 레코드들이 재로깅된다(유사코드의 15 ∼ 26번 문). 철회용 로그 레코드의 데이타는 부가적인 정보와 함께 ALR에 저장된다. 모든 ALR들은 ALR의 PrevLSN을 통해서 연결되어 있다. 제일 처음의 ALR의 PrevLSN은 값은 LSN_NIL 값을 가짐으로써 트랜잭션의 첫 ALR임을 나타낸다.

3.2.2 트랜잭션 철회

알고리즘 6에 제시한 트랜잭션 철회 루틴은 현재 수행중인 트랜잭션을 특정 SaveLSN까지 취소한다. 이 루틴에 대한 입력은 철회될 위치를 지정하는 SaveLSN과 철회될 트랜잭션을 지정하는 TransID이다. 이 단계에서는 로그레코드를 쓰여진 역순서로 철회하고, 철회된 로그레코드마다 CLR

[4]버퍼 풀 변경 페이지 테이블의 recLSN 값 중 최소값이다.
[5]트랜잭션 테이블의 firstLSN 값 중 최소값이다.
[6]검사점 연산 수행시 쓰여진 'BEGIN_CHKPT'로그레코드의 LSN값이다.
[7]데이타베이스 시스템 관리자가 이 값을 조절할 수 있다.
[8]성능을 위해서 이 경우 로그 레코드의 헤더 부분만 읽을 수 있다.

Algorithm 1 RELOGGING(TransTable, truncLSN)

```
1  트랜잭션 테이블에서 수행중인 트랜잭션의 firstLSN과 recoveryLSN의 차를 계산한 UndoOverhead(T)가
   특정 한계치보다 큰 트랜잭션을 선택하여 재로깅 트랜잭션 테이블을 만든다. 재로깅 트랜잭션 테이블은
   transID와 RelogNxtLSN으로 이루어져 있다.
   ▷ 재로깅 트랜잭션 테이블(RelogTable)의 초기 RelogNxtLSN값은 트랜잭션 테이블의 undoNxtLSN값으로 초기화 되
   어있다
   ▷ 재로깅 트랜잭션 테이블에 있는 각 장기간 수행되는 트랜잭션에 대해 역방향으로 읽어서 재로깅할 로그 레코드를 분
   석한다
2  while (재로깅 트랜잭션 테이블에 유효한 RelogNxtLSN이 존재하면) do
3      RelogNxtLSN ← 재로깅 트랜잭션 테이블로부터 가장 큰 RelogNxtLSN을 선택;
4      if (RelogNxtLSN < recoveryLSN) then do
           ▷ 재로깅 될 로그들이 이미 재로깅 된 로그 일 경우 RelogNxtLSN을 보정해주는 역할을 한다
5          RelogTable[TransID].RelogNxtLSN ← TransTable[TransID].lastALR;
6          continue;
7      end if ;
8      LogRec ← Log_Read(RelogNxtLSN);
9      if (RelogNxtLSN < truncLSN or LogRec.Type == 'ALTERNATIVE') then
10         if (Undo 가능한 로그 레코드이면) then LSN_stack에 LSN을 저장;
11     if (LogRec.Type == 'COMPENSATE') then
12         RelogTable[LogRec.TransID].RelogNxtLSN ← LogRec.undoNxtLSN;
13     else RelogTable[LogRec.TransID].RelogNxtLSN ← LogRec.prevLSN;
14 end while ;
   ▷ LSN_stack으로부터 LSN을 꺼내어 재로깅 한다
15 while ( RelogNxtLSN ← Get_LSN_Stack() ) do
16     LogRec ← Log_Read(RelogNxtLSN);
       ▷ 검사점 연산 중 재로깅 실시 중에도 트랜잭션이 완료하거나 취소 될 수 있으므로 트랜잭션 테이블에서 조사하여
       이미 완료되거나 취소된 트랜잭션인 경우에는 재로깅 작업을 중단한다
17     if ((nextTrans = transTable.find(LogRec.TransID)) == 0) then continue;
       ▷ 읽은 로그 레코드를 재로깅한다
18     AltLogRec.Type ← 'ALTERNATIVE';
19     AltLogRec.originLSN ← RelogNxtLSN;
20     AltLogRec의 모든 항목을 채워 놓고, undo-only 타입으로 set한다
21     if (RelogNxtLSN == nextTrans.firstLSN or RelogNxtLSN == nextTrans.firstALR) then
           AltLogRec.prevLSN ← LSN_NIL;
22     else AltLogRec.prevLSN ← nextTrans.lastALR;
23     Log_Write(AltLogRec, newLSN);
24     nextTrans.lastALR ← newLSN;
25     if (RelogNxtLSN == nextTrans.firstLSN and RelogNxtLSN == nextTrans.firstALR) then
           nextTrans.firstALR ← newLSN;
26 end while ;
```

그림 5: 재로깅 알고리즘

Algorithm 2 ROLLBACK(SaveLSN, TransID)

```
1  UndoNxt ← transTable[TableID].undoNxtLSN;              ▷ 철회할 첫번째 로그레코드의 주소
2  if ( UndoNxt.isvalid() and UndoNxt < RecoveryLSN ) then
       UndoNxt = TransTable[TableID].lastALR;
3  LogRec ← Log_Read(UndoNxt);                             ▷ 처리할 레코드를 읽는다
4  if (로그 레코드 타입이 'ALTERNATIVE'이면) then CompareLSN ← LogRec.originLSN;
5  else CompareLSN ← UndoNxt;
6  while (CompareLSN.isvalid() and SaveLSN < CompareLSN) do
7      switch ( LogRec.Type ) do
8          case ('UPDATE'or 'ALTERNATIVE') do
9              if (Undo 가능한 로그레코드이면) then do
10                 if (대체로그레코드이고 CompareLSN >= TransTrable[TransID].undoNxtLSN) then
                   continue;
11                 Page ← fix&latch(LogRec.PageID, 'X');
12                 Undo_Update(Page.LogRec);
13                 Log_Write('COMPENSATE', LogRec.TransID, transTable[TransID].lastLSN,
                       LogRec.PageID, LogRec.prevLSN, ...,
                       LgLSN.data);                        ▷ write CLR
14                 Page.LSN ← LgLSN;
15                 TransTable[TransID].lastLSN ← LgLSN;
16                 unfix&unlatch(Page);
17             end if ;
18             if (로그 타입이 'ALTERNATIVE'인 경우) do
19                 TransTable[TransID].lastALR ← LogRec.prevLSN;
20             end if ;
21             UndoNxt ← LogRec.prevLSN;
22         end case ;                                      ▷ Case('UPDATE'이거나 'ALTERNATIVE'인 경우)
23         case ('COMPENSATE')                             ▷ CLR인 경우 UndoNxtLSN 값만 읽는다
24             UndoNxt ← LogRec.undoNxtLSN;
25         elsewhere UndoNxt ← LogRec.prevLSN;             ▷ 레코드를 무시
26     end switch ;
27     TransTable[TransID].undoNxtLSN ← UndoNxt;
28     if ( UndoNxt.isvalid() and UndoNxt < RecoveryLSN ) then
           UndoNxt = TransTable[TableID].lastALR;
29     LogRec ← Log_Read(UndoNxt);                         ▷ 처리할 레코드를 읽는다
30     if (로그 레코드 타입이 'ALTERNATIVE'이면) then
           CompareLSN ← LogRec.originLSN;
31     else CompareLSN ← UndoNxt;
32 end while ;
```

그림 6: 트랜잭션 철회 알고리즘

이 쓰여진다. 철회되어야 할 로그레코드('UPDATE'혹은 'ALTERNATIVE')를 만나면 이를 처리하고 난 후, 다음에 처리할 로그레코드는 로그레코드의 PrevLSN필드를 검사함으로써 결정된다. 재수행용 로그레코드는 철회단계에서는 무시한다.

다음에 처리할 로그레코드가 이미 재로깅되었다면 트랜잭션 테이블 엔트리에 있는 LastALR(T)을 참조하여 다음 처리할 로그레코드를 결정한다(2 번, 28 번 문). 로그 레코드의 Type필드가 'ALTERNATIVE'인 경우, 그 레코드의 OrigLSN이 SaveLSN과 비교된다. 트랜잭션이 철회되는 동안에도 재로깅 루틴에 의해서 일부 로그 레코드가 재로깅될 수 있으므로 OrigLSN값이 트랜잭션 테이블 엔트리의 UndoNxtLSN보다 적은 ALR 로그레코드에 대해서만 취소해야 한다(10번 문).

그림 4를 예로 보자. 만약 트랜잭션 $T1$이 취소된다면, 철회 루틴은 먼저 트랜잭션 테이블의 UndoNxtLSN에 의해서 가르켜진 로그레코드 6부터 처리하게 된다. 로그레코드 6과 5를 철회하고 난 후, 다음으로 처리할 로그레코드 4의 LSN이 현재 RecoveryLSN보다 적다는 걸 알 수 있다. 그러므로 LastALR(T)에 의해 가르켜진 로그레코드 $A2$가 다음에 처리할 로그 레코드가 된다. 다음으로 로그레코드 $A1$이 처리된다. 모든 대체로그레코드들은 PrevLSN필드에 의해서 연결되어 있다.

3.3 재시동 고장 회복

3.3.1 분석 단계

고장 회복 수행동안 ARIES는 가장 최근의 검사점으로부터 고장 직전에 로그에 저장된 로그레코드까지 검사하는 분석단계를 가진다. 여기서 검사점 시작점으로부터 고장이 일어나기 바로 직전 상황까지 수행되던 트랜잭션에 대한 분석과 변경 페이지 테이블에 대해 분석한다. 이 단계를 통해서 고장 당시의 모든 수행중인 트랜잭션에 대한 정보와 변경 페이지에 대한 정보를 재구성할 수 있다. 이 루틴에 대한 입력 값으로는 가장 최근의 검사점 연산의 시작점을 가르키는 매스터 로그 레코드의 LSN이 사용된다. 그리고 이 루틴을 수행시키고 나면, 트랜잭션 테이블과 변경 테이블이 결과 값으로 반환된다. 이 루틴의 전반적인 처리과정은 기존 ARIES와 유사하기 때문에 이 루틴의 유사 코드는 생략했다.

재시동 고장분석 단계에서 읽혀진 로그레코드가 'END_CHKPT'인 경우 트랜잭션 테이블과 변경 페이지 테이블은 그 안의 정보를 반영하기 위해 수정된다. 대체로그레코드에 관련된 모든 정보는 로그를 검색하면서 얻어지는 게 아니라 'END_CHKPT'로그 레코드내에 있는 해당 정보를 읽음으로써 얻어진다. 재로깅은 검사점 연산 수행시에만 일어나고, 재로깅이 완료되었는지 아닌지에 대한 결정은 'END_CHKPT'로그레코드에 디스크에 기록되었는지로 단일하게 결정되기 때문이다. 그러므로 고장회복 분석 단계에서는 'ALTERNATIVE'로그레코드를 무시한다.

3.3.2 재수행 단계

재시동 고장회복의 두번째 단계는 재수행 단계이다. 이 단계는 ARIES처럼 이력의 재수행(repeating history)을 담당한다. 해당 페이지에 반영되지 않은 모든 갱신은 재반영된다. ARIES에서와 같이 철회된 트랜잭션에 의한 갱신이라도 재수행된다. 대체 로그레코드는 철회용이기 때문에 ARIES의 재수행 알고리즘은 수정없이 사용될 수 있다.

3.3.3 철회 단계

재시동 고장회복의 세번째 단계는 철회 단계이다. 철회 과정을 구현한 RESTART_UNDO 루틴이 그림 7에 기술되어 있다. 이 루틴에 대한 입력은 트랜잭션 테이블이다. 이 루틴은 기존의 ARIES와 유사하다. 아직 완전하게 철회되지 않은 미완료 트랜잭션 중에서 가장 큰 UndoNxtLSN값을 가진 로그레코드를 다음 처리할 로그레코드로 선택한다. 이 과정은 트랜잭션 테이블에 진행할 어느 트랜잭션도 남아있지 않을 때까지 수행된다. 이 루틴에서 트랜잭션 T에 해당하는 엔트리의 UndoNxtLSN 필드 값이 RecoveryLSN보다 적은 경우 UndoNxtLSN필드를 LastALR(T)값으로 바꾼다(2 번 문). 왜냐하면 가장 최근에 재로깅된 로그 레코드가 트랜잭션 테이블에 있는 LastALR(T)값에 의해서 가르켜지기 때문이다.

고장시에 ARIES/RL을 이용한 재시동 고장회복의 예가 그림 8에 설명되어 있다. 모든 로그레코드는 같은 페이지에 대한 갱신을 나타낸다. 실패하기 전에 페이지는 두번의 갱신을 페이지에 했다. 그 후 부분철회를 수행해 로그레코드 4와 3을 철회하였다. 그 결과로 CLR $4'$와 $3'$가 로그에 기록되었다. 트랜잭션은 계속 수행이 되고, 검사점 연산에 의해 재로깅이 일어났다. 검사점 연산에서는 RecoveryLSN전의 갱신이 되지 않은 모든 재수행용 로그레코드(3, 4, $4'$, $3'$)가 디스크에 반영이 되고 3.2.1에 설명된 재로깅 알고리즘에 의해 철회용 로그레코드(1과 2)가 재로깅된다. 그래서 대체로그레코드인 $A1$과 $A2$가 생성되었다. 여기서 고장이 일어났다고 하자. 먼저 ARIES/RL은 재시동 고장회복 재수행 단계를 통해 디스크에 반영되지 않은 로그레코드 5와 6을 디스크에 반영시킨다. 그리고 철회용 로그레코드 6, 5, $A2$와 $A1$이 철회된다. 로그레코드 5의 PrevLSN필드 값이 RecoveryLSN보다 적기 때문에 LastALR(T)에 의해서 가르켜진 $A2$가 철회되게 된다.

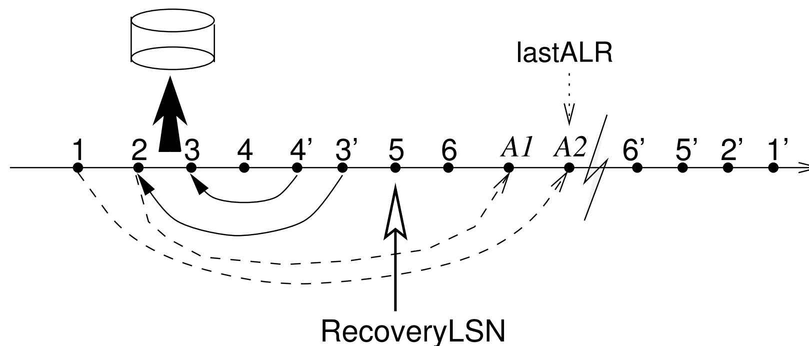
4 ARIES/RL의 평가

이 장에서는 본 논문에서 제안한 ARIES/RL에 대한 성능평가를 보인다. ARIES/RL은 서울대학교에서 개발된 SRP(**S**NU **R**elational DBMS **P**latform)[1, 21]위에 구현되었다. SRP는 C++을

Algorithm 3 RESTART_UNDO(TransTable)

```
1  while EXISTS(Trans with State = 'U' in TransTable) do
2      UndoLSN ← maximum(UndoNxtLSN) from TransTable entries with State = 'U';
       ▷ 취소 되어야 하는 가장 큰 UndoNxtLSN값을 가진 트랜잭션을 선택한다. 가장 큰 UndoNxtLSN값을 찾을 때
       찾은 값이 RecoveryLSN보다 적은 경우에는,해당 UndoNxtLSN값을 트랜잭션 엔트리의 LastALR(T)값으로 변
       경한다
3      LogRec ← Log_Read(UndoLSN);                    ▷ 다음 로그레코드를 읽는다
4      switch (LogRec.Type) do
5          case ('UPDATE') do
6              if (취소가능한 로그레고드이면) then do
7                  로그레코드를 처리하고, CLR을 기록한다;
8              end if ;                               ▷ 취소가능로그레코드인경우
9              else TransTable[LogRec.TransID].undoNxtLSN ← LogRec.prevLSN;
               ▷ 재수행용 로그레코드인 경우에는 무시한다
10             if (LogRec.prevLSN = 0) then do
                   ▷ 취소가 끝난 경우, 'end' 로그 레코드를 기록한다
11                 Log_Write('end', LogRec.TransID, TransTable[LogRec.TransID].lastLSN, ...);
12                 delete TransTable entry where TransID ← LogRec.TransID;
13             end if ;                               ▷ 트랜잭션이 이미 완료가 된 경우
14         end case ;
           ▷ 다음 분석할 로그 레코드를 찾는다
15         case ('COMPENSATE')
16             TransTable[LogRec.TransID].undoNxtLSN ← LogRec.undoNxtLSN;
17         case ('ROLLBACK'or 'PREPARE')
18             TransTable[LogRec.TransID].undoNxtLSN ← LogRec.prevLSN;
19     end switch ;
20 end while ;
```

그림 7: 재시동 고장회복시에 트랜잭션 철회 알고리즘



Redo	5	6		
Undo	6	5	2	1

그림 8: 재시동 고장회복의 예

이용해서 구현된, ARIES를 지원하는 클라이언트-서버 관계형 DBMS이다. 성능 평가는 SparcStation 20에서 이루어졌다.

재로깅에 관여된 항목들만을 평가하기 위해서 시간과 관계된 기준을 사용하지 않았다. 대신 ARIES/RL을 두가지 방법으로 평가하였다. 첫째 여러 트랜잭션이 수행되고 있을 때 제한된 로그 공간에서 장기간 트랜잭션이 최대 얼마만큼의 갱신을 할 수 있는지를 조사하였다. 이를 통해서 기존의 ARIES에 비하여 ARIES/RL이 장기간 트랜잭션이 존재하는 경우 얼마나 더 효율적인지를 조사할 수 있다. 둘째 검사점 연산 수행시에 재로깅을 함으로써 일어나는 부가적인 부하에 대해서 조사하였다.

4.1 평가 모델

평가 모델은 저장시스템, 트랜잭션 관리자, 로그 관리자, 그리고 고장 회복 관리자로 구성되어 있다. 트랜잭션 관리자는 트랜잭션을 만들고 관리한다. 트랜잭션이 한 페이지에 갱신을 하면 그에 대한 재수행/철회(redo/undo)용 로그 레코드가 생성되어 저장시스템을 통해서 로그 디스크에 저장된다.

먼저 평가를 위해 사용된 여러 변수에 대해 설명한다. 로그 디스크의 크기는 40 페이지로 구성되어 있다[9]. 한 페이지는 8K바이트 크기로 이루어져 있다. 검사점 연산은 로그 공간의 12%가 찼을 때마다 수행이 된다[10]. 로그에 로그 데이타가 추가될 때 사용되는 로그용 버퍼는 12 페이지이다. 그리고 트랜잭션을 철회하거나 재로깅할 때 사용되는 읽기 버퍼는 2 페이지이다. 로그 레코드의 총 크기는 444 바이트이다. 이는 44 바이트의 로그 헤더와 400 바이트의 데이타 부분(재수행용 200 바이트, 철회용 200 바이트)으로 구성되어 있다. 모든 갱신용 로그레코드는 재수행/철회용 데이타를 가지고 있다. 성능 평가를 간단하게 하기 위해, 모든 재수행용/철회용 데이타들은 물리적 로그 데이타의 형태로 디스크에 저장되는 것으로 가정하였다. 갱신된 데이타의 양이 200 바이트이므로, 각각 재수행용/철회용 로그데이타의 크기는 각각 200 바이트씩이다. 대체 로그레코드의 크기는 244 바이트이다. 대체 로그레코드는 로그헤더와 철회용 로그 데이타부분으로 구성되어 있다. 짧은 트랜잭션(short-duration transaction: SDT)의 갯수는 임의로 5개에서 10개 사이로 잡았다. SDT는 LDT보다 페이지를 10배정도 자주 갱신한다. 재로깅 한계치는 전체 로그 공간의 30%로 잡았다. 만약 한 트랜잭션의 UndoOverhead(T)가 이 한계치보다 크게 된다면, 이 트랜잭션은 장기간 트랜잭션으로 간주되어 재로깅의 대상이 된다.

[9]보통 실제 DBMS에서 로그 디스크의 크기는 이것보다는 크겠으나, 여기서는 평가의 편의를 위해서 40 페이지로 잡았다. 우리는 이 값이 성능평가의 일반성을 깨뜨리지는 않으리라 믿는다.

[10]검사점 연산의 주기는 시간으로 잡을 수도 있고, 이처럼 로그 공간의 사용양으로 잡을 수도 있다.

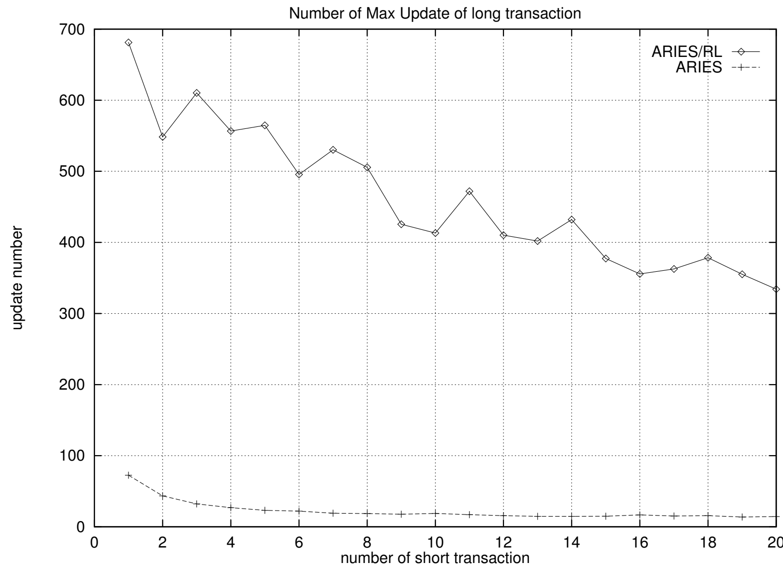


그림 9: 장기간 트랜잭션에 의한 최대 갱신 수

4.2 평가 결과

그림 9는 장기간 트랜잭션에 의한 최대 갱신치를 나타낸다. 최대 갱신치는 로그 공간이 다 차서 트랜잭션을 철회하지 않고서는 더이상의 트랜잭션의 진행이 불가능할 때까지의 갱신수를 조사하였다. 이 수치는 SDT의 수를 각각 1부터 20까지 변화시키면서 조사하였다. 각 결과는 10번의 테스트를 통해 나온 평균값이다. 그리고 RedoLSN은 버퍼를 강제로 디스크에 기록하는 과정을 통해 'BEGIN_CHKPT'로그 레코드의 LSN과 같게 조절되었다. SDT의 갯수가 둘일때, ARIES기법을 사용한 LDT는 평균적으로 페이지를 43.5번 갱신하였다. ARIES/RL을 사용했을 때에는 LDT는 평균적으로 548.4번의 갱신을 하였다. 같은 크기의 로그 공간을 가지고 수행했을 경우 ARIES/RL 기법이 기존 ARIES 알고리즘에 비해 더 많은 갱신을 할 수 있다는 것을 나타낸다. 갱신 작업을 통해 생성된 로그레코드의 갯수는 동일하기 때문에 ARIES/RL이 로그 공간을 더 효율적으로 사용했음을 보여주고 있다. SDT의 갯수가 많을 수록 LDT의 페이지 갱신 수는 줄어든다. 이는 SDT가 재로깅 중에도 계속 진행해 로그에 로그레코드를 추가하기 때문이다. LDT의 최대 갱신 수가 그림 9에 있는 그래프의 수치보다 적은 경우에 주어진 로그 공간에서 그 트랜잭션을 완료될 수 있다.

그림 10은 온라인 로그 공간에 남아 있어야 하는 로그 공간의 크기와 재로깅에 의해서 발생하는 부하를 나타낸다. X축은 검사점 연산 수행 횟수를 나타내고, Y축은 남아 있어야 하는 공간을 전체 로그 공간 크기에 대한 백분율로 나타내었다. 이 평가는 SDT가 일곱개, LDT의 갯수가 한개인 상태에서 조사되었다. 먼저 장기간 트랜잭션 T의 UndoOverhead(T)에 대해서 조사하자. UndoOverhead(T) 내에 있는 로그 레코드들은 트랜잭션 철회를 위하여 온라인 로그 공간에 남아 있어야 한다. 이 값은

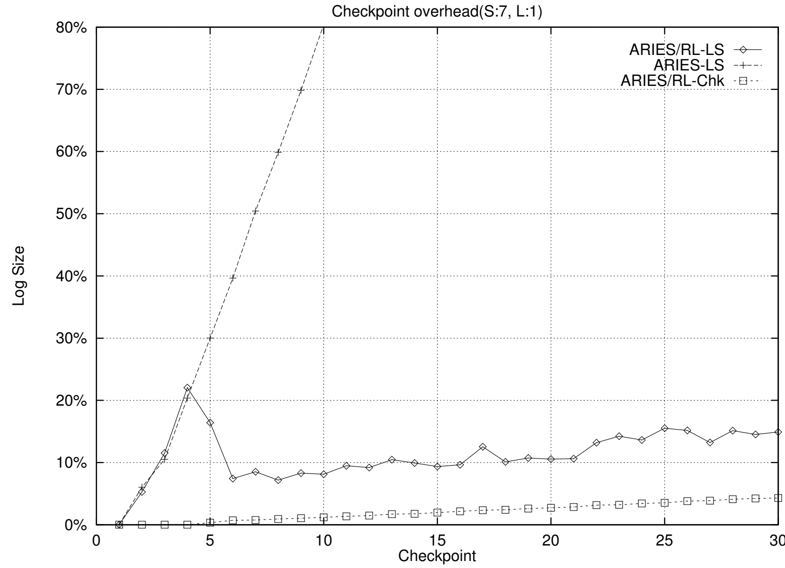


그림 10: 검사점 연산시의 온라인상 유지해야 될 로그 공간 크기와 검사점 설치 부하

그림 10에서 ARIES/RL-LS과 ARIES-LS 그래프로 나타나 있다. 여기서 ARIES의 그래프는 로그 공간을 축약시키는 것을 막고있는 LDT 때문에 온라인 로그 공간에 남아 있어야 하는 로그의 양이 계속 증가하는 것을 볼 수 있다. 반면 ARIES/RL-LS의 경우는 처음에는 ARIES-LS의 그래프와 비슷하게 증가하다가 ARIES/RL-LS의 값이 재로깅 한계치로 지정된 30%보다 큰 경우, 재로깅이 수행되어 줄어드는 것을 알 수 있다. 재로깅은 검사점 연산 수행중에 시행되는데, 재로깅될 로그 레코드의 크기가 많을 수록, 검사점 연산에 더 많은 시간이 걸린다. ARIES/RL-Chk 그래프는 검사점 연산 수행 시간을 나타낸다. 이 검사점 연산 수행 시간을 'BEGIN_CHKPT'과 'END_CHKPT'로그 레코드 사이의 LSN 차이[11] 로 살펴보았다. ARIES의 경우에는 수행되는 트랜잭션의 수나 상태에 따라 달라질 수 있겠으나 거의 고정 값인 1%이내 값을 유지하고 있다. 검사점 연산 수행시 ARIES/RL에서는 부가적으로 재로깅 작업이 수행됨으로 재로깅할 로그레코드의 수가 많아지면 전체적인 검사점 연산 수행 시간이 길어진다는 것을 볼 수 있다. 그러나 재로깅될 로그 레코드가 없는 경우에는 검사점 연산의 수행 시간은 기존의 ARIES와 동일하다. 그리고 ARIES/RL에서 재로깅을 하는데 너무 많은 시간이 걸린다면 재로깅을 유발한 트랜잭션을 취소시킬 수 있다.

그림 11에는 검사점 연산 완료시에 조사된 장기간 트랜잭션 T의 UndoOverhead(T)가 나타나있다. ARIES에서는 LDT가 처음 로그 레코드를 쓰고는 아직 완료하거나 취소되지 않아 UndoLSN이 항상 같은 값을 유지하기 때문에 UndoOverhead(T)값은 계속 증가한다. 그래프에서 ARIES-undo로

[11] 이 값은 현재 진행중인 트랜잭션에 의해서 달라질 수 있다. 알고리즘에서 검사점 연산이 트랜잭션의 진행을 막지 않기 때문이다.

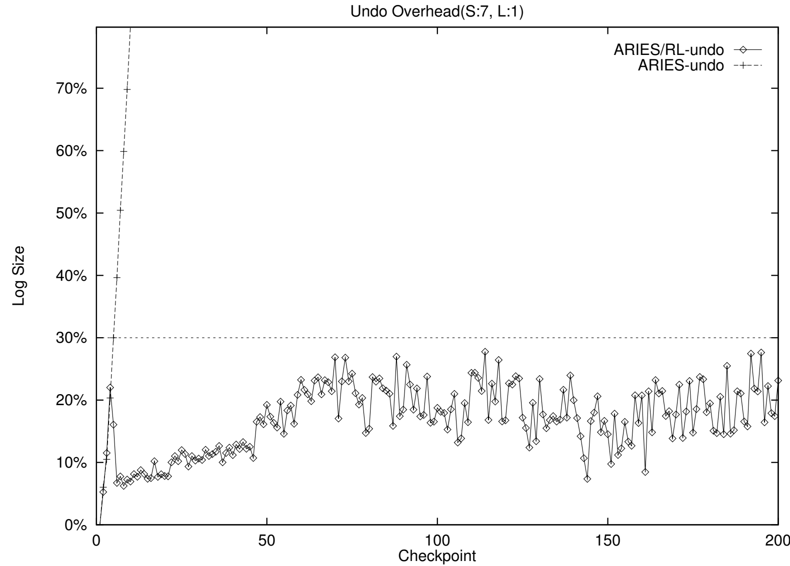


그림 11: 검사점 연산시의 LDT T의 UndoOverhead(T)

표시되어 있다. 재로깅을 하는 경우에는, UndoOverhead(T)가 한계치로 정한 30%를 넘어가는 경우 재로깅이 수행되어 이 값은 항상 한계치인 30%보다 적게 된다. 그래프에서 `ARIES/RL-undo`로 표시되어 있다.

ARIES/RL은 재로깅을 통하여 장기간 트랜잭션이 존재하는 경우, 로그 공간을 효율적으로 사용하게 한다. 반면 재로깅이 검사점 수행시간에 이루어지므로 재로깅될 로그레코드 갯수가 많아짐에 따라서 검사점 수행시간이 길어질 수 있다. 하지만 위에서 언급했듯이, 과도하게 검사점 수행시간을 소요하게 만드는 트랜잭션에 대해서는 기존 방법과 동일한 방법을 취하도록 동적으로 결정할 수 있기 때문에 합리적인 검사점 수행 시간을 얻을 수 있다.

5 관련 연구

장기간 트랜잭션이 존재하는 경우 제한된 로그 공간의 낮은 효율성을 해결하기 위한 노력들이 있어 왔다[3, 7, 8, 11, 13, 14, 16, 17, 19]. 이 장에서는 이에 대한 관련 연구를 설명하고, ARIES/RL와의 차이점에 대해 설명하겠다.

[11]에는 *동적 로깅(dynamic logging)*이라고 불리는 기법이 제시되었다. LDT가 존재하는 경우 온라인 로그 공간의 낮은 효율성 문제를 개선하기 위해서 *복사용 로그(copy aside log)*라고 불리는 특별한 화일을 만들어서 예전 로그들을 옮기는 기법이다. 다른 기법으로는 분리된 로그 공간을 사용하지 않고 로그 레코드를 로그의 앞쪽으로 옮기는 기법이 있다. 이것은 Camelot 트랜잭션 관리자[7, 8]에서

사용되었고, *전위 복사(copy forwarding)*라고 불리고 있다[11]. 또한 Hagmann과 Garcia-Molina는 [14]에서 장기간 트랜잭션에 의해 야기되는 디스크 관리 문제를 해결하기 위해 로그 레코드를 순환하는 방법을 제시하였다. 그러나 이러한 단순한 전위 복사(copy forwarding)기법을 ARIES에 적용하는 것은 문제를 가지고 있다. ARIES알고리즘은 연산로깅(operation logging)을 지원하고 신축적인 공간관리를 가능하게 한다. 그리고 부분철회와 중첩최상위 연산(nested top action)이 가능하다. 이 중첩 최상위 연산을 이용하여 트랜잭션이 취소되더라도 트랜잭션이 행한 일부 작업이 완료할 수 있게 된다[20]. ARIES는 CLR에 의해서 여러 장점을 가지고 있는데, 이 CLR을 고려하지 않은 전위 복사 방법은 데이타베이스의 일관성을 깨뜨릴 수 있다. 이 논문에서는 고장회복 기법으로 널리 사용되고 있는 ARIES에 전위 복사기법을 적용하여 장기간 트랜잭션을 지원하도록 하였다.

로그 압축(Log Compression) 기법[13]은 로그를 저장하기 위한 공간을 절약하기 위해 불필요한 정보를 필터링한다. 그러나, 기존의 로그 압축 기법은 로그 압축을 위하여 '배치 작업'이 요구되는 문제점이 있다. 로그 레코드들이 압축되는 동안에는 새로운 로그 레코드들이 기존의 로그에 추가될 수 없다.

로그에서처럼 메모리 공간 상에서 메모리 활용을 높이려는 연구가 있어 왔다[19]. 메인 메모리 공간을 여러개의 임시적인 세대로 분할하여 시스템의 쓰레기 수집(garbage collection) 작업이 최근 세대(younger generation)에서만 일어나도록 하였다. 제안된 방법은 프로그래밍 언어의 데이터 객체 접근 패턴에 의존하여 공간을 관리한다. 그러나 이 의존성이 로그 레코드들 사이의 의존성보다 복잡하기 때문에 이 방법을 직접 로그를 위한 디스크 공간 관리 기법으로 사용하기엔 부적절하다[17].

John S. Kenn과 William J. Dally는 *단명로깅(ephemeral logging)*방법을 제안하였다[16, 17]. 단명로깅(EL)은 로그를 새로운 레코드들이 계속 추가되는 큐의 체인으로 관리한다. 그리고 계속적으로 쓰레기 수집과 로그 압축을 수행한다. 로그상에 남아있어야 하는 로그에 대해서만 큐[12]의 앞부분에서부터 다음 큐의 끝부분으로 옮겨지게 된다. 이 방법은 재수행만을 사용하는 고장회복 알고리즘에만 적용될 수 있다. 그리고 마지막 세대(generation)의 로그 레코드들은 계속 옮겨져야 하기 때문에 ARIES/RL보다 빈번한 로그 I/O가 일어난다. 또한 연산 로깅을 사용할 때 단명 로깅을 사용하는 방법에 대한 시험적인 결과가 제시되었으나[15], 아직 많은 부분들에 대해 추가적인 연구가 필요하다.

ORACLE7에서는 트랜잭션 취소나 재시동 고장회복을 위하여 하나 이상의 취소용 세그먼트를 사용한다[3]. ORACLE7의 세그멘트는 테이블공간(tablespace)내에 위치한, 특정한 타입의 논리적 저장 구조를 가지는 데이타를 모두 포함하는 익스텐드(extent)의 집합이다. ORACLE7에서 데이타

[12] 이것을 세대(generation)이라고 부른다.

베이스의 취소용 세그멘트는 진행중인 트랜잭션에 의해 변화된 데이타의 예전 값을 가지고 있다. 한 트랜잭션은 하나의 취소용 세그멘트에 데이타를 기록한다. 트랜잭션이 장기간 지속되어 익스텐드가 부족하게 되면 ORACLE7은 같은 취소용 세그멘트의 가용 익스텐트를 찾는다. 이 때, 이미 취소용 세그멘트에 할당된 익스텐드를 재사용하거나 새로운 익스텐드를 할당받아서 사용한다. 그러나 이 가변 크기의 취소용 세그멘트는 구현하고 관리하기에 너무 복잡해지는 문제가 있다.

6 결론

이 논문에서 장기간 트랜잭션이 존재하는 경우 제한된 로그 공간을 효율적으로 사용하기 위하여, ARIES에 재로깅 기법을 이용하여 확장한 ARIES/RL 알고리즘을 제시하였다. 재로깅은 트랜잭션 취소나 재시동 고장회복시에 사용된 로그 레코드들은 로그의 앞쪽으로 옮기는 기법이다. 이 알고리즘은 기존의 ARIES알고리즘이 가지는 장점을 해치지 않을 뿐더러, 장기간 트랜잭션을 수행하는 경우 로그 공간을 더 효율적으로 사용하게 한다. 많은 장기간 트랜잭션들이 존재하는 경우 ARIES/RL에서의 검사점 연산 수행 시간이 길어질 수 있다. 하지만 데이타베이스 시스템 관리자가 UndoOverhead(T)의 한계치(threshold)와 재로깅의 사용여부를 조절할 수 있기 때문에, 어느 정도 합당한 검사점 연산 수행기간을 얻을 수 있다. 그리고 시스템에서도 데이타베이스의 상태에 따라서 동적으로 재로깅의 사용여부를 조절할 수 있다. 재로깅이 너무 많은 시간이 걸린다면, 재로깅을 유발하는 트랜잭션을 취소할 수 있다. 이 경우 기존의 ARIES알고리즘과 같은 검사점 연산 수행 시간을 갖는다.

앞으로 여러 변하는 환경속에서 시스템이 언제 재로깅을 수행할 것인지를 찾는 부분에 대한 연구가 필요하다. 그리고 현재 ARIES/RL 알고리즘에서는 구현의 편이를 위하여 재로깅이 검사점 연산 수행중에 일어나도록 되어있다. 이를 개선하여 재로깅이 임의의 시간에 수행될 수 있도록 알고리즘을 개선하는 연구가 필요하다.

참조 서적

[1] J. H. Ahn. "Object-Oriented Design of Extensible Storage System". Master's thesis, Dept. of Computer Engineering, Seoul National University, Feb. 1993. (in Korean).

[2] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. "ASSET: A System for Supporting Extended Transactions". In *Proc. of the ACM SIGMOD Conf. on Management*

of Data, pages 44–54, 1994.

[3] S. Bobrowski, editor. *ORACLE7 SERVER CONCEPTS MANUAL*. Oracle coperation, 1992.

[4] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita. "Data Caching Tradeoffs in Client-Server DBMS Architectures". In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 357–366, Denver, Colorado, May 1991. Also available as Technical Report #994, Comp. Sci. Dept., University of Wisconsin-Madison, Jan. 1991.

[5] P. Y. Chang and W. W. Myre. "OS/2 EE Database Manager overview and technical highlights". *IBM Syst. J.*, 27(2):105–118, 1988.

[6] P. K. Chrysanthis and K. Ramamritham. "Synthesis of Extended Transaction Models Using ACTA". *ACM Trans. Database Syst.*, 19(3):450–491, 1994.

[7] D. Duchamp, J. J. Bloch, J. L. Eppinger, A. Z. Spector, and D. Thompson. "Design Rationale of the Camelot Distributed Transaction Facility". Technical Report CUCS-008-90, University of Columbia, 1990.

[8] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Data Management Systems. Morgan Kaufmann Publichers, Inc., San Mateo, CA, 1991.

[9] H. Garcia-Molina. "Sagas". In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 249–259, 1987.

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

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman Publishers, Inc., 1993.

[12] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. "Starburst Mid-Flight: As the Dust Clears". *IEEE Trans. on Knowledge and Database Eng.*, 2(1):143–160, Mar. 1990.

[13] R. B. Hagmann. "A Crash Recovery Scheme for a Memory-Resident Database System". *IEEE Transactions on Computers*, C-35(9), Sept. 1986.

[14] R. B. Hagmann and H. Gracia-Molina. "Implementing long lived transactions using log record forwarding". Technical Report CSL-91-2, Xerox Palo Alto Research Center, Feb 1991.

[15] J. S. Keen. Logging and recovery in a highly concurrent database. Technical Report AITR-1492, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, June 1994.

[16] J. S. Keen and W. J. Dally. "Performance Evaluation of Ephemeral Logging". In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Washington, DC, USA, May 1993.

[17] J. S. Keen and W. J. Dally. "Extended Ephemeral Logging: Log Storage Management for Applications with Long Lived Transactions". *ACM Trans. Database Syst.*, 22(1):1–42, Mar. 1997.

[18] T. J. Lehman and M. J. Carey. "A Recovery Algorithm for A High-Performance Memory-Resident Database System". In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 104–117, 1987.

[19] H. Lieberman and C. Hewitt. "A Real-Time Garbage Collector Based on Lifetimes of Objects". *Communications of the ACM*, 26(6):419–429, June 1983.

[20] 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.

[21] Y. J. Song. "Object-Oriented Design and Implementation of SQL Processor". Master's thesis, Dept. of Computer Engineering, Seoul National University, Feb. 1994. (in Korean).