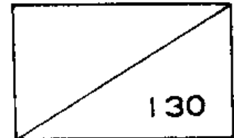
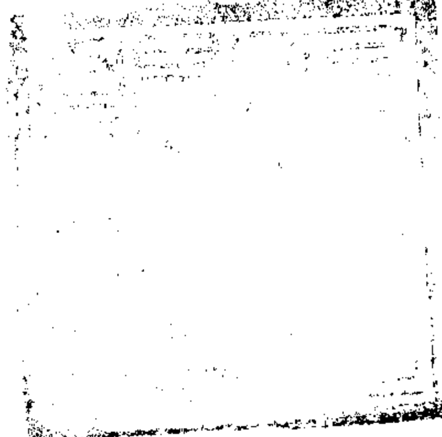


UNIX를 기반으로 한 데이타베이스 관리체제의 프로토타입 설계

Design of A Prototypical Database Management System Based on UNIX

연구기관
한 국 과 학 기 술 원



과 학 기 술 처

제 출 문

과학기술처 장관 귀하

본 보고서를 "기본 소프트웨어 기술연구"에 관한 세부 과제인 "UNIX를 기반으로 한 데이타베이스 관리체제의 프로토타입 설계"에 관한 최종 보고서로 제출합니다.

1990년 5월 30일

수행연구기관 : 한국과학기술원

연구책임자 : 문 송천 (한국과학기술원 전산학과 조교수)

연 구 원 : 임 종태 (한국과학기술원 전산학과 박사과정)

신 동천 (한국과학기술원 전산학과 박사과정)

안 종길 (한국과학기술원 전산학과 박사과정)

이 윤숙 (한국과학기술원 전산학과 박사과정)

김 유성 (한국과학기술원 전산학과 박사과정)

요 약 문

1. 제목

UNIX를 기반으로 한 데이타베이스 관리체제의 프로토타입 설계

2. 연구 개발의 목적 및 중요성

UNIX가 널리 보급됨에 따라 국내에서도 UNIX를 기반으로 하는 데이타베이스 관리체제의 개발이 요청되고 있다. 그러나 아직도 국내에서는 데이타베이스를 자체 설계하여 개발해 본 경험이 매우 부족한 실정이다. 본 연구에서는 UNIX를 기반으로 한 데이타베이스 관리체제의 프로토 타입을 개발함으로써 실제 시스템에 운영되도록 할 수 있을 뿐만 아니라 고급 인력의 배출과 데이타베이스 관리체제 기술의 발전에도 크게 기여할 수 있을 것이다.

3. 연구 개발의 내용 및 범위

데이타베이스 관리체제의 구성요소를 결정하고 각 구성 요소의 기능과 구성 요소들 사이의 인터페이스를 설계한 후 구현한다. 본 데이타베이스 관리체제는 질의 처리 부분과 트랜잭션 처리 부분으로 구성된다. 질의 처리 부은 사용자의 질의를 받아들여 최적화한 후 실행하는 부분이다. 트랜잭션 처리 부분은 트랜잭션의 동시성을 유지하고 고장으로부터의 회복시켜주는 부분이다. 질의 처리 부분에서는 사용가능한 버퍼를 최대한 이용하는 죠인 방법과 최적화 방법을 통해 성능을 향상시켰다. 트랜잭션 처리 부분에서는 성능 분석을 통해 여러 동시성 제어 기법, 교착 상태 탐지 기법, 회복 기법들 중에서 우수한 방법들을 선택하였다. 트랜잭션 처리의 하부 구조를 이루는 저장 시스템은 다중키와 다중 사용자, 로깅 등을 지원할 수 있도록 WISS(WIssconsin Storage System)를 확장하여 KASS (Kaist Storage System)라고 명명하여 사용하였다.

4. 연구 결과 및 활용에 관한 건의

본 연구에서는 관계형 데이타베이스 관리체제의 각 구성 요소를 정의하고 설계한 후 구현하였다. 이 연구 내용은 국내 자체 기술에 의한 데이타베이스 관리 체제의

설계로서 최초의 연구 결과가 될 것이다. 또한 UNIX를 기반을 한 데이타베이스 관리체제로서 교육용 데이타베이스 관리체제 및 국가 기간 전산망에 활용될 수 있을 것이다.

SUMMARY

Relational database systems have been attracted from users and researchers for the simple concept and easy interfaces. But there is no relational database management system that was originally designed and implemented in Korea. In this research project we design and implement a relational database management system on a workstation running under UNIX. For the design of the storage system, we extended the Wissconsin storage system to support multi-attribute keys and multi-users. This prototypical DBMS provides relational query language SQL to users.

CONTENTS

1. Introduction 1

2. Process Structure 4

2.1 Related works on Process Structure 4

2.2 Process Structure of developed DBMS 6

2.3 Communication between Processes 9

3. Query Processing 16

3.1 Overview of Query Processing System 17

3.2 Syntax Analysis and Transformation into Relational Algebra 19

3.3 Query Optimization 29

3.4 Implementation of Relational Operations 35

3.5 Implementation of Data Manipulation Language 41

3.6 Implementation of Catalog management System 42

4. Transaction Scheduler 51

4.1 Overview of Transaction Scheduler 51

4.2 Predicate Locking 54

4.3 Deadlock Detection 55

4.4 Implementation of Transaction Scheduler Program 58

5. Data and Recovery Manager 74

5.1 Overview of Data and Recovery Manager 74

5.2 recovery Algorithm 75

5.3 Implementation of Data and Recovery Manager 88

6. Conclusion 97

References 98

Appendix 1. SQL of SQL 103

Appendix 2. Regular Expression for Lexical Analysis 108

Appendix 3. Input of YACC 111

목 차

제 1 장 서 론 1

제 2 장 프로세스 구조 4

제 1 절 프로세스 구조에 대한 관련 연구 4

제 2 절 개발한 시스템의 프로세스 구조 6

제 3 절 프로세스들간의 통신 9

제 3 장 질의 처리 시스템 16

제 1 절 질의 처리 시스템의 개요 17

제 2 절 구문 분석과 관계 대수로의 변환 19

제 3 절 최적화 방법 29

제 4 절 관계 대수 연산자의 구현 35

제 5 절 질의를 제외한 데이타 조작어 구현 41

제 6 절 캐털로그 관리 구현 42

제 4 장 트랜잭션 스케쥴러 51

제 1 절 기본 개념 51

제 2 절 술어 로킹 54

제 3 절 상충성 검사 알고리즘 55

제 4 절 동시성 제어 프로그램의 구현 58

제 5 장 데이타 회복 관리 74

제 1 절 고장의 형태 74

제 2 절 고장 회복 방법들 75

제 3 절 개발한 시스템에서의 회복 관리 88

제 6 장 결 론 97

참고 문헌 98

부록 1. SQL 질의 구문 103

부록 2. SQL 어휘 분석용 정규 표현식 108

부록 3. YACC의 입력 111

제 1 장 서 론

경제 활동이 신장되고 사회가 발전함에 따라 점점 더 많은 정보가 생성되고 이 정보들을 효율적으로 관리 이용할 필요성이 증대되었다. 많은 양의 정보를 컴퓨터에 저장되고 저장된 정보의 정확성을 유지하며, 사용자의 요구에 대해 신속하게 처리해 주기위해 데이타베이스(database)를 사용한다. 지난 20년 동안 이러한 데이타베이스를 관리하고 운영하는 데이타베이스 관리체제(database management system; 약칭 DBMS)에 대한 연구가 활발히 진행되었다. 관계형 데이타베이스 관리체제의 주된 목표는 데이타베이스 내의 정보를 검색하거나 데이타베이스에 정보를 저장하는 데 있어서 편리하고도 효율적인 환경을 제공하는 것이다. 특히, 관계형 데이타베이스의 이론과 기술이 발전함에 따라 관계형 DBMS의 연구가 활발하다.

데이타베이스 관리체제는 기반으로 하고 있는 모델에 따라 망형과 계층형, 관계형 등으로 구분된다. 이 중 관계형 데이타베이스 관리체제는 계층형 데이타베이스나 망형 데이타베이스 등에 비하여 구조가 간단하고 사용하기 편리할 뿐만 아니라 사용자에게 다양한 편의를 제공한다. 따라서 정보 처리의 효율을 높이기 위해서 관계형 데이타베이스 관리체제의 이용은 계속 확산되고 있다.[Date 82],[Date 87]).

데이타베이스 관리 체제는 사용자의 요구를 들어주기 위하여 데이타베이스가 기반으로 하고 있는 운영체제가 가지는 기능을 일부 이용한다. 즉, 데이타베이스 관리체제는 데이타를 관리, 운영하고 동시에 일어나는 다수의 사용자의 요구를 만족시켜주기 위해서 컴퓨터에 있는 운체제의 버퍼 관리 기능, 화일 시스템, 동시성 제어 및 회복 기능을 이용할 수 있다. 따라서 데이타베이스 관리체제의 기능과 서능은 데이타베이스가 기반으로 하는 운영체제의 기능과 성능에 상당히 의존한다.

운영체제 중 UNIX는 처음에는 대학이나 연구기관 등에서 널리 쓰였으나 최근에

는 일반 산업기관에도 그 이용이 점점 확산되고 있다. 초기에 UNIX하에서 개발된 데이타베이스 관리체제로는 Ingres와 Troll이 널리 알려져 있다. 이 두 시스템은 모두 UNIX의 화일체제를 이용하여 구현되었다. UNIX의 화일체제는 큰 화일을 저장함에 있어서 실제 디스크의 연속적인 블럭에 배당하지 못하기 때문에 순차 액세스를 할 때 탐색 시간(seek time)이 커지게 된다. 따라서 위 두 시스템 모두 대용량의 데이타베이스 체제에서는 성능이 나쁘다. 그 뒤 Ingres를 상용화할 때 UNIX의 화일체제를 이용하지 않고 원시 디스크(raw disk)를 직접 액세스하여 성능을 개선하였다.

캘리포니아 주립 대학 Berkely 분교(U.C. Berkely)에서 Ingres를 개발할 때와 거의 비슷한 시기에 IBM에서는 System R을 개발하였다. 그 뒤 System R의 질의어인 SQL을 구현한 여러 관계형 데이타베이스 관리체제들 (예, ORACLE, DB/2, informix, Unify)이 발표되었다. 이들은 일반적으로 특정 운영체제를 기반으로 하지 않고 여러 운영체제에 이식하여 운영된다.

이러한 세계적인 추세에 발 맞추어 국내에서도 몇몇 기관에서 데이타베이스 관리체제를 개발하였으나 현재 상품화하여 널리 쓰이는 데이타베이스 관리체제는 거의 없는 실정이다. 국내의 대부분의 기과에서는 외국에서(주로 미국) 개발한 데이타베이스 관리체제를 수입하여 사용하고 있다. 한편 국내에서도 데이타베이스 기술의 이론과 기술에 대한 연구가 성숙되어 이제는 상용화할 수 있는 시점에 와 있다. 이러한 시점에서 본 연구실에서는 UNIX를 기반으로 하여 관계형 데이타베이스 관리 체제의 프로토타입(prototype)을 개발하고 구현한다. 본 연구에서 개발한 데이타베이스 관리 체제는 질의를 처리하는 부분과 트랜잭션의 정확성을 유지하는 부분으로 구성된다. 질의 처리 부분은 사용자의 SQL 질의를 구문 분석한 후 최적화하여 실행한다. 트랜잭션 처리 부분에서는 트랜잭션 단위로 동성 제어와 회복을 제공한다. 그리고 저장 시스템으로는 Wissconsin 대학에서 개발한 WIssconsin Storage System(WISS)를 본 연구실

에서 확장하여 KASS(KAIST Storage System) 라고 명명하여 사용하였다.

본 연구에서는 UNIX를 기반으로 한 관계형 데이타베이스 관리체제의 프로토타입을 개발하는 것이 주된 목표이다. 제 2 장에서는 본 연구에서 개발한 데이타베이스 관리체제의 프로세스 구조와 프로세스 사이의 통신에 대해 설명한다. 제 3 장에서는 SQL 질의를 대화식으로 처리하는 질의 처리 방법에 대해 기술한다. 제 4 장에서는 트랜잭션 관리와 동시성 제어 방법에 대하여 기술한다. 제 5 장에서는 데이타 관리와 고장 회복 방법에 대해 기술한다. 끝으로, 본 연구의 결론과 앞으로의 연구 방향을 제 6 장에서 기술한다.

제 2 장 프로세스 구조

이 장에서는 관계형 데이타베이스 관리 체제의 프로세스 구조중에서 대표적인 다중 서버 방식과 단일 서버 방식에 대해 기술하고 본 연구에서 개발한 데이타베이스 관리 체제의 프로세스 구조에 대해 기술한다.

제 1 절 프로세스 구조에 대한 관련 연구

다수의 사용자를 위한 데이타베이스 관리 체제의 구조는 일반적으로 사용자의 요구를 받아들여 분석하는 요청 프로세스(requestor process 또는 front-end interface)와 데이타베이스를 접근하여 요청을 처리해주는 서버 프로세스(server process 또는 back-end process)로 구성된다. 요청 프로세스는 사용자의 명령를 받아들여 구문 분석하고 최적화한 후 서버 프로세스를 호출하여 사용자의 명령을 실행한다. 서버 프로세스는 요청 프로세스로부터 명령을 받아서 트랜재션 단위의 정확성이 유지되도록 실행하여 결과를 요청 프로세스에게 전달한다.

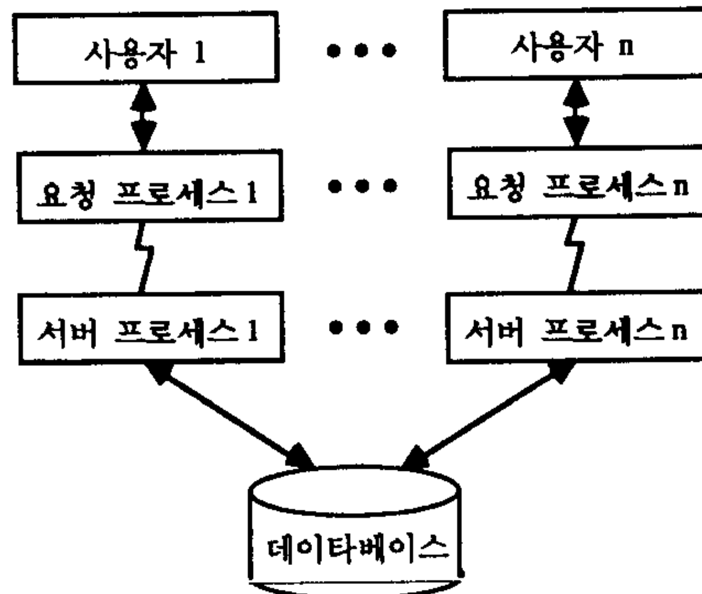


그림 2.1 다중 서버 방식의 프로세스 구조

다수 사용자를 지원하기 위한 데이타베이스 관리 체제를 구현하기 위한 프로세스 구조에는 각각의 요청 프로세스마다 하나의 서버 프로세스가 존재하는 다중 서버 방식(그림 2.1)과 데이타베이스 관리 체제내에 하나의 서버 프로세스만이 존재하는 단일 서버 방식(그림 2.2)이 있다.

다중 서버 방식의 프로세스 구조는 각 서버가 서로 다른 처리기(processor)에서 수행될 수 있으므로 다중 처리기 시스템(multiprocessor system)에 적합하고, 또한 각 사용자마다 하나의 서버 프로세스를 가지게 되므로 병목 현상 없이 사용자의 요청을 처리해 줄 수 있다. 그러나 단일 처리기 시스템에서는 프로세스의 수가 크게 증가하여 프로세스 스위칭하는 부가 노력이 커져서 전체 시스템의 성능을 저하 시킬 수 있다. 또한 각 프로세스간의 통신을 하기 위한 부가 노력도 함께 커짐으로 단일 처리기 시스템에는 적합하지 않다.([Kim 87][Rod 89])

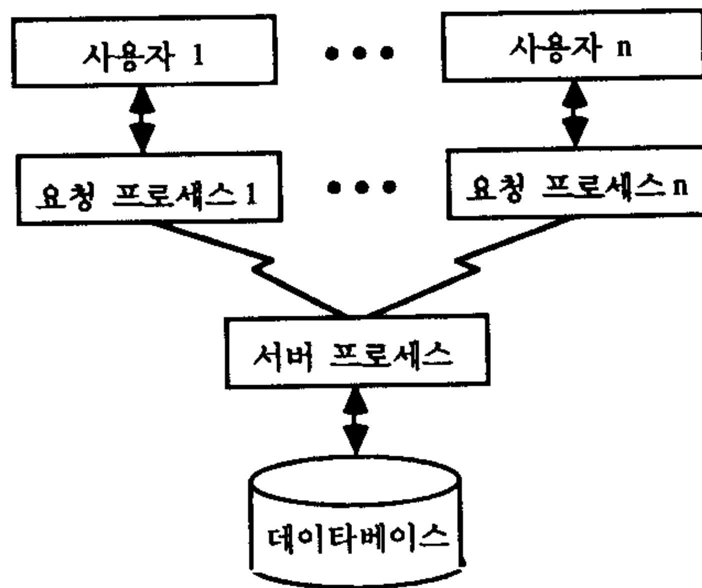


그림 2.2 단일 서버 방식의 프로세스 구조

단일 서버 방식의 프로세스 구조는 데이타베이스 관리 체제를 위한 프로세스의 수가 크게 증가하지 않으므로 단일 처리기 시스템에서 프로세스 스위칭을 위한 부가

노력을 줄일 수 있다. 그러나 하나의 서버 프로세스가 여러 요청 프로세스를 서비스 해주게 됨으로써 서버 프로세스에 부하가 집중되고 부하 집중으로 인한 병목현상이 발생할 수 있는 문제점을 가지고 있다.

제 2 절 개발한 시스템의 프로세스 구조

본 연구에서는 다수의 사용자를 지원하기 위한 데이타베이스 관리 체제의 프로세스 구조로 집중 방식을 확장한 프로세스 구조를 채택하였다. 데이타베이스 관리 체제 전체에 서버 기능은 하나가 존재하지만 이를 하나의 프로세스로 구현하지 않고 기능적인 분업을 할 수 있는 두 개의 독립적인 프로세스로 구현하였다. 따라서 데이타베이스 관리 체제에서 트랜잭션 처리 시스템을 위한 프로세스는 트랜잭션의 동시성을 제어하는 프로세스와 데이타베이스를 액세스하며 회복기능을 제공하는 프로세스로 구성된다. 이러한 구조는 단일 처리기 시스템에서 프로세스의 개수을 증가하지 않으며 프로세스간의 통신을 위한 부가 노력을 줄일 수 있는 구조이며, 또한, 데이타베이스 관리 체제를 구현함에 있어 기능별 모듈로 전체 시스템을 구성할 수 있는 장점을 가지고 있다.

본 연구에서 개발한 데이타베이스 관리 체제의 전체 프로세스 구조는 그림 2.3과 같다. 그림에서 타원은 UNIX의 프로세스를 나타내며, 사각형은 메세지 큐(message queue) 혹은 공유 메모리(shared memory)를 나타낸다.

본 연구에서 개발한 데이타베이스 관리 체제의 전체 프로세스 구조는 질의 처리 시스템(Query Processing System)과 트랜잭션 처리 시스템(Transaction Processing System)으로 구성된다. 질의 처리 시스템은 각 사용자의 SQL 명령을 입력 받아 구문 분석하여 최적화한 후 트랜잭션 처리 시스템을 호출하여 실행한다.

트랜잭션 처리 시스템은 트랜잭션 스케쥴러(Transaction SCHeduler: 약어 TSCH)와 데이타 회복 관리 프로그램(Data and Recovery Manager: 약어 DRM)으로 구성된다. 트랜잭션 스케쥴러는 질의 처리 시스템으로부터 제기되는 트랜잭션에 대한 동시성을 제어하여 데이타베이스의 일관성(consistency)을 유지시켜 준다. 데이타 회복 관리 프로그램은 버퍼와 물리적 저장장치를 운영하여 트랜잭션의 원자성(atomicity)을 유지한다. 따라서 트랜잭션 처리 시스템은 사용자가 제기한 트랜잭션에 대한 원자성을 보장해 주며, 데이타베이스의 일관성을 유지시켜 준다.

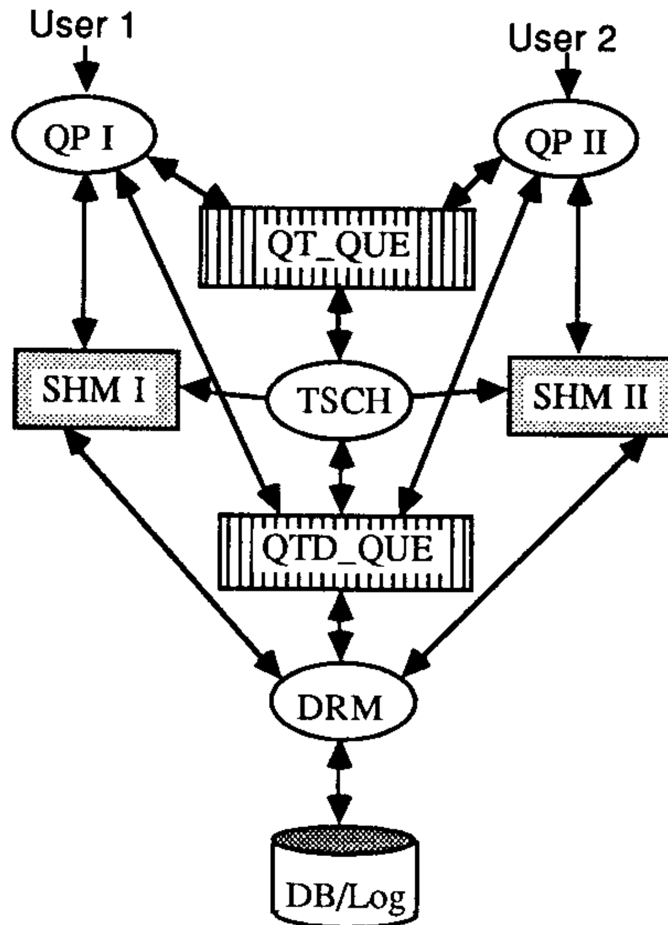


그림 2.3 트랜잭션 처리 시스템의 구성

본 연구에서 트랜잭션 처리 시스템을 위한 기능을 두 개의 프로세스로 구현한 이유는 전술한 바와 같이 프로세스의 개수를 크게 증가시키지 않는 범위내에서 기능별로 프로세스를 구현하여 전체 시스템을 모듈화하기 위한 구조이다. 또 트랜잭션 스케쥴러에서 각 명령어의 동시성을 미리 검사하여 지연해야 할 명령을 실행전에 검사함으로써 데이타 회복 관리 프로그램내에서는 각 명령이 실행하다 중지되는 상황은 발생하지 않는다.

사용자가 제기한 트랜잭션은 질의 처리 프로그램을 통하여 트랜잭션 스케쥴러에게 전달된다. 트랜잭션 스케쥴러는 트랜잭션의 제기에서부터 트랜잭션의 실행 완료시까지 트랜잭션의 실행을 관리한다. 따라서 트랜잭션 스케쥴러는 질의 처리 프로그램으로부터 트랜잭션의 연산을 메세지 큐를 통하여 전달받아서 이를 처리한 후 메세지 큐를 통하여 데이타 회복 관리 프로그램에게 전달한다. 데이타 회복 관리 프로그램은 트랜잭션 관리 프로그램으로부터 전달 받은 트랜잭션의 연산에 대해서 회복에 필요한 준비 작업을 하고 저장 시스템을 호출하여 연산을 실행한다.

본 연구에서는 데이타 회복 관리 프로그램에 병목 현상이 생겨 트랜잭션 처리 시스템 전체의 성능을 저하시키는 경우를 방지하기 위해 데이타 회복 관리 프로그램의 시스템 우선 순위(system priority)를 변화시키는 방법을 사용하였다. 즉, 하나의 사용자가 데이타베이스 관리 체제를 이용할 때에는 데이타 회복 관리 프로그램의 시스템 우선 순위가 다른 프로세스의 시스템 우선 순위와 같게 하여 작동시킨다. 그러나 두명 이상의 사용자가 데이타베이스 관리 체제를 사용할 때에는 하나의 데이타 회복 관리 프로그램이 모든 동시 사용자의 질의 처리 프로그램의 요청을 서비스해주어야 하므로 데이타 회복 관리 프로그램에 많은 부하가 걸린다. 따라서 동시 사용자의 수가 2이상일 때에는 데이타 회복 관리 프로그램의 시스템 우선 순위를 높여서 다른 프로세스보다 상대적으로 빠르게 작동하게 함으로서 데이타 회복 관리 프로그램

에서 예상되는 병목 현상을 제거하였다. 그러나 트랜잭션 스케쥴러는 데이타 회복 관리 프로그램과 달리 사용자의 모든 명령를 처리하는 것이 아니라 동시성 제어가 필요한 명령만을 전달 받아서 처리하므로 데이타 회복 관리 프로그램보다 부하가 심하지 않으므로 트랜잭션 스케쥴러에는 이런한 기능을 부여하지 않았다. 본 시스템의 동시성 제어 방법으로는 술어 로킹(Predicate Locking) 방법을 사용하기 때문에 스캔(scan) 명령내에서의 레코드 액세스에 대해서는 동시성 제어가 필요하지 않다.

다수 사용자의 질의 처리 프로그램에 대해 서비스해 주기 위해 질의 처리 프로그램, 트랜잭션 스케쥴러, 그리고 데이타 회복 관리 프로그램들은 독립적인 프로세스로 구현되어 트랜잭션을 정확히 처리하기 위해서 각 명령 혹은 명령을 처리하기 위한 정보 그리고 명령의 실행 결과를 교환해야 한다. 독립적으로 구현된 프로세스들간의 통신 방법에 관해서는 3 절에서 자세히 논의한다. 질의 처리 프로그램과 트랜잭션 스케쥴러와 데이타 회복 관리 프로그램등대해서는 각각 제 3장과 제 4장, 그리고 제 5장등에서 논의한다.

제 3 절 프로세스들간의 통신

본 연구에서 개발된 데이타베이스 관리 체제에서는 트랜잭션 스케쥴러와 데이타 회복 관리 프로그램은 각기 다른 독립적인 UNIX 프로세스로 구현되어 데몬(demon) 프로세스의 형태로 시스템내에 항시 동작 상태로 존재하게 된다. 또한, 각 사용자의 질의를 처리하기 위한 질의 처리 프로그램은 각 사용자가 운영체제의 쉘(shell)에서 명령어 "sql"을 입력했을 때 생성되며 명령어 "quit"을 입력했을 때 소멸된다. 이 질의 처리 프로그램은 사용자의 질의를 받아 번역하고 이를 내부의 레코드 단위 명령으로 변환하여 트랜잭션 스케쥴러에게 전달하여 준다. 독립적인 프로세스들간의 통신을 위해서는 SYSTEM V의 프로세스간의 통신(InterProcess Communication: 약어

IPC) [Bac 86] 기능중에서 메세지 기능(message function)과 공유 메모리 기능 (shared memory function)을 사용한다.

본 연구에서 메세지 기능은 프로세스들간에 길이가 정해진 정보 혹은 명령을 전송할 때 사용하고, 공유 메모리 기능은 가변 길이의 정보 혹은 명령의 실행 결과를 레코드 단위로 전송할 때 사용한다. 이와 같이 두 방법을 사용한 이유는 다음과 같다. 만일 메세지 기능으로 모든 정보를 전송한다면, 길이가 가변적인 정보를 전달하기 위해서는 최대 길이로 메세지 형태를 정의해서 이를 이용하여 메세지를 보내게 됨으로써 메모리를 낭비하게 된다. 또한, 공유 메모리 기능으로 모든 정보를 전송한다면, 공유 메모리에 대한 동시 액세스를 방지하기 위한 배타적 액세스 기능과 통신하는 두 프로세스간의 통신 내용 분석에 대한 프로토콜이 추가로 필요하게 된다. 이러한 기능을 SYSTEM V의 IPC 기능중에서 세마포어(semaphore) 기능을 사용하여 구현할 수 있으나, 본 연구에서는 구현의 간편함을 위해 SYSTEM V의 IPC 기능중 메세지 기능을 이용하여 이를 해결하였다.

2.3.1 메세지 기능

질의 처리 프로그램은 사용자의 질의를 내부의 레코드 단위 연산으로 변환하여 메세지 형태로 트랜잭션 스케쥴러 혹은 데이타 회복 관리 프로그램에게 전달한다. 트랜잭션의 연산중에서 동시성 제어를 위한 연산들은 스케쥴러에게 전달되고, 동시성 제어를 하지 않아도 되는 연산들은 데이타 회복 관리 프로그램에게 직접 전달된다. 본 연구에서 구현한 트랜잭션 처리 시스템에서 동시성 제어 기법으로는 술어 로킹을 사용하므로 데이타를 액세스하기 위한 모든 명령에 대해 동시성 제어를 할 필요가 없고 릴레이션에 대한 액세스를 시도하기 위한 술어를 기술한 명령들에만 동시성 제어를 하면된다. 따라서 릴레이션을 액세스하기 위한 술어를 포함하는 명령어(이들의 종류는 동시성 제어 방법을 설명하는 장에서 자세히 기술함.)와 트랜잭션을 관리하기

위한 명령(예를 들면, TBEGIN, COMMIT, ABORT 등), 그리고 프로세스의 정보를 교환하기 위한 명령(예를 들면, START, EXIT 등)은 트랜잭션 스케쥴러에게 전달되고 이를 제외한 모든 명령들은 데이타 회복 관리 프로그램에게 직접 전달된다.

명령의 실행 결과는 데이타 회복 관리 프로그램에서 질의 처리 프로그램으로 직접 전달된다. 그림 2.3에서 QT_QUE 큐는 질의 처리 프로그램과 트랜잭션 스케쥴러와의 메세지 교환을 위한 메세지 큐이고 QTD_QUE 큐는 질의 처리 프로그램과 트랜잭션 스케쥴러 그리고 데이타 회복 관리 프로그램 사이의 메세지 교환을 위한 메세지 큐이다. 메세지 큐를 생성하고 운용하는 명령은 다음과 같다.

가. 메세지 큐의 생성

독립적인 프로세스가 다른 프로세스와 메세지 통신을 하기 위해서는 메세지 큐를 먼저 생성해야 한다. 메세지 큐를 생성하기 위한 명령의 형태는 다음과 같다.

```
msgqid = msgget(key,flag)
```

명령의 형태에서 msgqid는 메세지 큐의 식별자로서 msgget 명령어의 실행 결과로 받게 된다. 또한, key는 메세지 큐를 지정하기 위한 필드로서 다른 두 프로세스들이 같은 메세지 큐를 사용하기 위해서는 같은 key를 사용해서 메세지 큐를 생성해야 한다. 그림 2.3에서 트랜잭션 스케쥴러는 질의 처리 시스템과 통신하기 위해 메세지 큐 QT_QUE를 사용한다. 또한 트랜잭션 스케쥴러는 데이타 회복 관리 프로그램과 통신하기 위해 메세지 큐 QTD_QUE_ID를 사용한다. 따라서 트랜잭션 스케쥴러는 통신하기 전에 QT_QUE_ID 큐와 QTD_QUE_ID 큐를 생성해야 한다. 이를 위한 명령들은 다음과 같다.

```
#define    QT_QUE_KEY     1
#define    QTD_QUE_KEY    2
```

```
QT_QUE = msgget(QT_QUE_KEY, IPC_CREATE|0777);
QTD_QUE = msgget(QTD_QUE_KEY, IPC_CREATE|0777);
```

생성된 메세지 큐를 소멸시킬 때에는 msgctl 명령를 사용한다. 명령 msgctl의 형식은 다음과 같다.

```
msgctl(msgqid, IPC_RMID, 0);
```

위의 명령에서 msgqid는 msgget 명령으로 큐를 생성할 때 부여한 큐 식별자(예: QT_QUE 혹은 QTD_QUE)이고 IPC_RMID는 메세지 큐를 소멸하라는 명령 코드이다.

나. 메세지의 전송

각 프로세스간에 통신을 하기 위한 메세지 형태는 다음과 같다.

```
#define    MAX_LEN_MESG    128;
struct    msgform    {
            int    mtype;
            char  mtext[MAX_LEN_MESG];
}
```

메세지를 위한 데이타 구조 msgform은 메세지의 형태를 분류하기 위한 mtype 필드와 메세지 내용을 포함하는 mtext 부분으로 구성된다. 메세지 형태를 표시하는 mtype 필드는 메세지 큐에 있는 여러 메세지 중에서 자신에게 오는 메세지만을 전달받기 위해 사용한다. 즉, 트랜잭션 스케쥴러는 QTD_QUE 큐에 있는 여러 메세지중에서 mtype의 값이 1인 메세지만을 전달받는다. 또한, 데이타 회복 관리 프로그램은 QTD_QUE 큐에서 mtype 값이 2인 메세지만을 전달받는다. 따라서 트랜잭션 스케쥴러가 데이타 회복 관리 프로그램에게 메세지를 보낼 때에는 메세지의 mtype 부분을 2로 만들어 보내면 다른 프로세스는 이를 액세스 하지 못하고 데이타 회복 관리 프로그

램만이 이를 액세스하게 된다.

각 프로세스에서 메세지를 전송할 때 다음과 같은 명령을 사용한다. 메세지의 내용이 msg 데이타 구조에 있을 때 이를 보내기 위해서는 msgsnd 명령을 사용한다. msgsnd 명령의 형식은 다음과 같다.

```
msgsnd(msgqid, msg, length, flag);
```

또한, 메세지 큐로부터 하나의 메세지를 꺼내올 때에는 msgrcv 명령를 사용한다. 명령 msgrcv의 형식은 다음과 같다.

```
msgrcv(msgqid, msg, length, type, flag);
```

위의 명령어들에서 msgqid는 msgget 명령어를 이용해서 생성된 메세지 큐의 식별자를 나타내고, msg는 msgsnd 명령어에서는 전송할 메세지의 내용을, msgrcv 명령어에서는 수신할 메세지를 보관할 자료 구조를 나타낸다. 파라메타 length는 메세지의 길이를 나타내며, flag는 메세지 전송과정에서 오류가 발생할 때 이를 처리하기 위한 방법을 표시한다. 그림 2.3에서 트랜잭션 스케쥴러가 데이타 회복 관리 프로그램에게 메세지를 전달하기 위해서는 아래와 같은 방법을 사용한다.

```
#define    TO_DRM    2

msg.mtype = TO_DRM;
msgsnd(QTD_QUE, &msg, sizeof(msgform), 0);
```

데이타 회복 관리 프로그램이 트랜잭션 스케쥴러가 보낸 위의 메세지를 받기 위해서는 다음과 같은 방법을 사용한다.

```
#define    TO_DRM    2

msgrcv(QTD_QUE, &msg, sizeof(msgform), TO_DRM, 0);
```

따라서 데이타 회복 관리 프로그램은 QTD_QUE 큐에서 mtype 값이 2인 메세지만을 전달받는다.

2.3.2 공유 메모리 기능

프로세스들이 가변 길이의 정보 혹은 명령의 실행결과를 전송할 때에는 공유 메모리를 사용한다. 공유 메모리는 질의 처리 프로그램이 불리어질 때마다 생성되어 이를 질의 처리 프로그램, 트랜잭션 스케쥴러 그리고 데이타 회복 관리 프로그램이 공유하게 된다. 공유 메모리를 생성하기 위해서는 shmget 명령를 사용한다. 명령 shmget의 형식은 다음과 같다.

```
shmid = shmget(key,size,flag);
```

변수 shmid는 공유 메모리의 식별자로서 shmget 명령의 실행결과이다. 파라메타 key는 공유 메모리를 지정하기 위한 필드로서 메세지 기능에서와 마찬가지로 여러 프로세스들이 같은 공유 메모리를 사용하기 위해서는 같은 key 값을 주어야 한다. 또한 size는 공유 메모리의 크기를 나타낸다. 명령 shmget을 이용하여 생성된 공유 메모리를 프로세스의 가상 메모리로 사용하기 위해서는 shmat 명령를 사용한다. 즉, 프로세스는 생성된 공유 메모리를 가상 메모리로 사용하기 위해서는 공유 메모리를 가상 메모리의 일부분으로 첨가하여야 한다. 명령 shmat의 형식은 다음과 같다.

```
virtadrs = shmat(shmid, addr, flag);
```

위의 명령으로 생성된 공유 메모리를 소멸시킬 때에는 shmctl 명령를 사용한다. 명령 shmctl의 형식은 다음과 같다.

```
shmctl(shmid, IPC_RMID, 0);
```

위의 명령들에서 shmid는 shmget 명령어를 이용하여 생성된 공유 메모리의 식별자를 나타내며, addr는 공유 메모리가 프로세스의 가상 메모리로서 첨가되기를 원하는 가상 메모리 주소를 나타낸다.

제 3 장 질의 처리 시스템

데이타베이스 관리 체제(DBMS)의 높은 성능을 달성하기 위해서는 효율적인 질의 처리 방법의 개발이 필수적이다. 본 연구에서는 데이타베이스 언어를 새로 설계하지 않고 1987년에 ISO에서 국제 표준으로 정한 SQL [ISO 87]을 사용하였다. SQL의 구문 형식은 부록 1에 수록되어 있다. SQL은 언어 형태면에서 INGRES와 대비되는데, 구문이 단순하여 기억하기 쉽다는 점과 INGRES의 튜플 변수 (tuple variable) 개념을 잘 이해하지 못하는 보통 사용자들이 쉽게 사용할 수 있다는 두가지 점에서 장점을 갖고 있다. 그러나 구문 분석 프로그램이 같은 형태의 질의 트리를 생성해 주면 질의 시스템의 최적화 프로그램 이하는 구문 분석 프로그램의 영향을 받지 않으므로, 다른 질의어나 다른 형태의 인터페이스를 제공하기 위해서는 각각에 대해 구문 분석 프로그램만 작성하면 된다.

데이타베이스 언어는 크게 데이타 정의어 (data definition language: 약칭 DDL)와 데이타 조작어 (data manipulation language: 약칭 DML)로 대별된다. 질의어는 DML 중에서 데이타베이스로부터 자료를 찾아보는 부분만을 의미한다. SQL에서 질의는 SELECT문으로 표현된다. 즉 SQL에서 질의 처리는 SELECT문의 효율적인 처리로 압축된다. 일반적으로 질의는 다른 명령에 비해 자주 제기되며, DBMS의 성능을 나타내는 주요지표이다.

제 1 절 질의 처리 시스템의 개요

그림 3.1은 본 논문에서 설계하고 구현한 질의 처리 시스템의 전체적인 구성을 보여준다.

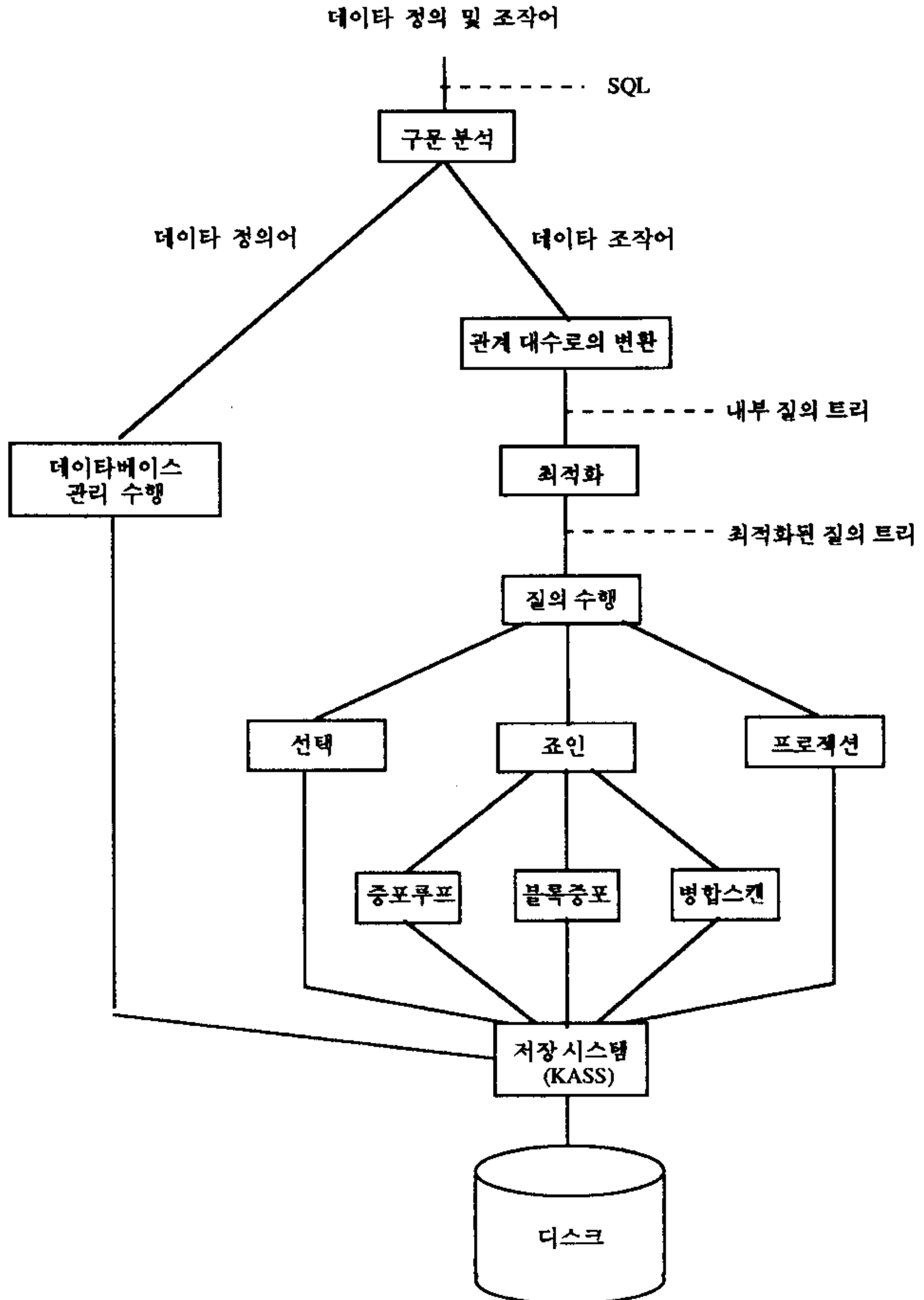


그림 3.1 질의 처리 시스템의 개요

본 질의 처리 시스템에서는 현재 SQL의 SELECT문 중에서 다음과 같은 형태의 질의만을 처리해 준다:

```
SELECT         attribute_list
[INTO          relation_name]
FROM           relation_list
[WHERE         boolean_expression]
[GROUP BY      attribute_list]
[HAVING        boolean_expression]
[ORDER BY      attribute_list]
```

위 질의 구문에서 "[ ]"는 각 질의에서 필요에 따라 없어도 되는 절을 표시한다. INTO 절의 relation_name은 이 질의의 수행 결과를 저장하는 릴레이션 이름을 의미한다. WHERE 절의 boolean_expression은 각 프레디킷트(predicate)를 AND, OR, NOT 등으로 연결한 식이다. 각 프레디킷트는 attribute_name <comparision_operator> value 형태로 구성된다. 여기서 <comparision_operator>는 다음과 같은 일곱 개의 비교 연산자로 구성된다: { EQ, NE, GT, GE, LT, LE, LIKE }. 여기서 주목할 점은 각 프레디킷트의 연산자로 질의 블록을 허용하지 않는다는 점이다. 즉, 본 연구에서 구현한 질의 처리 시스템은 비중포 SQL 질의를 처리하였다. 현재 중포된 SQL 질의와 커서 관련 연산은 구현하지 않았으며 WHERE절의 boolean_expression에 A + B와 같은 대수식과 통계함수는 허용되지 않는다. HAVING절의 boolean_expression은 대수식은 허용되지 않으나 통계함수는 허용된다. 현재 제공되지 않는 기능은 추후 확장될 예정이다.

그림 3.1에서 보는 바와 같이 구문 분석이 끝난 질의는 관계 대수로 변환된다. 그 이유는 대부분의 질의 최적화 방법 [Wong 76, Seli 79]들이 관계대수를 기반으로 하고 있기 때문이다. 본 시스템은 질의를 최적화하는데 사용하기 위한 내부 질의 표현 방법으로 관계 대수(Relational Algebra)를 사용한다. 현재까지의 많은 질의 최적화 방법들이 관계 대수로 표현되어 있다 [Wong 76, Seli 79]. 또한, 관계 대수는

연산자로서 릴레이션을 사용하여 튜플 변수(Tuple Vaiable)에 기초를 둔 관계 해석보다 분산 환경이나 데이타베이스 기계(Database Machine) 등에 적합한 장점을 가지고 있다. 최적화된 질의는 질의 실행 프로그램에 의해 선택, 조인, 프로젝션 등을 사용하여 수행된다. 이 중에서 조인이 가장 많은 실행 시간을 소요하는 연산이다. 본 시스템에서는 중포 루프와 블록 중포, 병합 스캔 등의 조인 방법들을 구현한다. 각각의 관계 대수 연산 프로그램들은 저장 시스템이 제공하는 인터페이스를 이용하여 구현한다.

제 2 절 구문 분석과 관계 대수로의 변환

구문 분석기는 Lex[Lesk 75]와 Yacc[John 78]을 이용하여 구현하였으며 데이타 조작어및 정의어를 사용자로부터 입력받아 번역하여 내부 질의 표현으로 변환한다. SQL 어휘의 정규 표현식과 SQL 구문의 BNF syntax는 각각 부록 2과 부록 3에 수록되어 있다. 각 명령어는 구문 분석 과정을 거치면 내부 질의 표현으로 변환된다. 구성된 내부 질의 구조는 관계 대수 변환 프로그램으로 입력된다. 본 시스템은 [Ceri 85]가 제안한 SQL을 관계 대수로 변환하는 방법을 참조하였다. SQL 질의를 입력받아 관계 대수로 변환시키는 변환 프로그램(Translator)은 그림 3.2와 같이 이름 변환 단계, 전처리 단계, 의미 추출 단계, 후처리 단계의 네 단계로 구성되어 있다.

이름 변환 단계는 SQL 질의를 입력받아 질의 내에 포함되어 있는 속성의 이름을 릴레이션 이름으로 확장시키고, 변수 이름도 릴레이션과 속성 이름으로 변환시킨다. 전처리는 SQL 질의를 관계 대수로의 변환을 쉽게 하기 위하여 일반적인 SQL 질의를 제한된 형태의 SQL 부질의로 표현한다. 의미 추출 단계는 제한된 형태의 SQL 부질의로부터 각 부질의에 대응되는 관계 대수 연산식을 구성한다. 후처리 단계는 구성된 관계 대수 연산식에서 필요없는 연산을 제거하고 중간 결과를 나타내는 연산식

들을 하나의 연산식으로 구성한다.

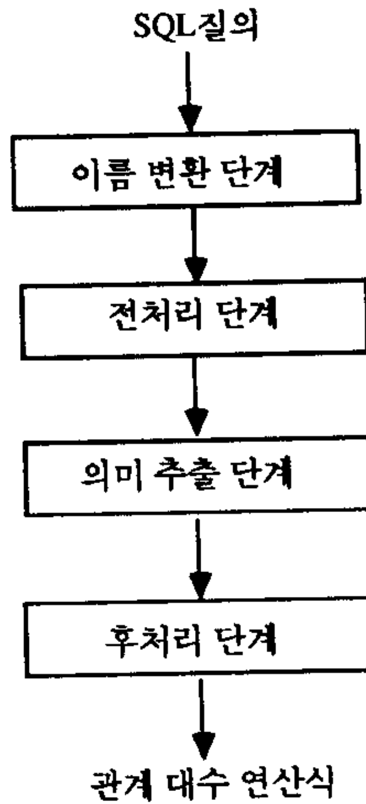


그림 3.2 관계 대수로의 변환

2.1 이름 변환 단계

(1) 같은 기본 릴레이션이지만 서로 다른 실례를 나타내는 것은 '(aphostrophy)를 사용한다.

```
SELECT A
FROM R
WHERE B = SELECT C FROM R

=> SELECT A
   FROM R'
   WHERE B = SELECT C FROM R"
```

(2) 속성 이름을 그 속성이 속해 있는 릴레이션으로 확장시킨다.

```
=> SELECT R'.A
     FROM R'
     WHERE R'.B = SELECT R''.C FROM R''
```

(3) SQL에서 FROM 절에서 릴레이션과 관계된 변수를 제거한다.

```
=> SELECT R'.A
     FROM R', R''
     WHERE R'.B = R''.B
```

2.2 전처리 단계

SQL 부질의 in, in-set, not, all, any, not-exist, contains, does-not-contain, set-equality, set-inequality, compound의 부질의를 다섯가지의 기본적인 SQL 부질의 simple, group-by, complex(nesting), exists, binary(union, difference, intersection)로 변환한다.

(1) In-query

In-query는 다음과 같이 "IN" 연산자를 "=" 연산자로 바꾼다.

```
SELECT <selector>
FROM <relation-list>
WHERE <left-term> IN <ngb-query>

=> SELECT Q.<selector>
     FROM Q.<relation-list>
     WHERE Q.<left-term> = Q.<ngb-query>
```

(2) In-set query

In-set query는 다음과 같이 "IN" 연산자를 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
```

```
WHERE (...(Q.<attribute_spec> = Q.<constant_1> OR
           Q.<attribute_spec> = Q.<constant_2> OR) ...
           Q.<attribute_spec> = Q.<constant_n>)
```

(3) Not-in query

Not-in query는 다음과 같이 "NOT IN" 연산자를 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
MINUS SELECT Q.<selector>
FROM Q.<relation_list>
WHERE Q.<left_term> = Q.<ngb_query>
```

(4) All query

All query는 다음과 같이 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
MINUS
SELECT Q.<selector>
FROM Q.<relation_list>
WHERE Q.<left_term> NOT (Q.<comp_op>) Q.<ngb_query>
```

(5) Any query

Any query는 다음과 같이 Any를 제거한다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
WHERE Q.<left_term> Q.<comp_op> Q.<ngb_query>
```

(6) Not-exists query

Not-exists query는 다음과 같이 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
MINUS
```

```
SELECT Q.<selector>
FROM Q.<relation_list>
WHERE Q.<exists_predicate>
```

(7) Contains query

Contains query, 즉 subquery Q_1이 subquery Q_2를 포함하는 질의는 다음과 같이 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
MINUS
SELECT Q.<selector>
FROM Q.<relation_list>
WHERE EXISTS (Q.<ngb_query_2>
      MUNUS Q.<ngb_query_1>
```

(8) Does-not-contain query

Does-not-contain query는 다음과 같이 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
WHERE EXISTS (Q.<ngb_query_2>
      MINUS Q.<ngb_query_1>
```

(9) Set-equality query

Set-equality query는 다음과 같이 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
WHERE (Q.<ngb_query_1) IN Q.<ngb_query_2> AND
       Q.<ngb_query_2) IN Q.<ngb_query_1>)
```

(10) Set-inequality query

Set-inequality query는 다음과 같이 변환시킨다.

```
SELECT Q.<selector>
FROM Q.<relation_list>
WHERE (Q.<ngb_query_1) NOT IN Q.<ngb_query_2> OR
      Q.<ngb_query_2) NOT IN Q.<ngb_query_1>)
```

(11) Compound query

Compound query는 다음과 같이 변환시킨다.

```
SELECT *
FROM Q.<relation_list>
WHERE Q.<compound_predicate>.1
T(Q.<compound_predicate>.<boolean>)
SELECT *
FROM Q.<relation_list>
WHERE Q.<compound_predicate>.3)

        where T(AND) = INTERSECT
              T(OR) = UNION
```

각 질의가 전처리 단계에서 5가지의 부질의로 변환하는 단계는 다음과 같이 표현될 수 있다.

i. in-subquery ---> complex subquery
ii. in-set subquery --> simple subquery
iii. not-in subquery --> binary subquery --> simple query, complex subquery
vi. all subquery --> binary subquery --> simple subquery, complex subquery
v. any subquery --> complex subquery
vi. not-exists sub. --> binary subquery --> simple subquery, complex subquery
vii. contains subquery --> binary subquery --> simple subquery,complex subquery
viii. Does-not-contain sub. --> exists subquery --> bunary subquery
ix. set-equality query --> compound subquery -->contains subquery, complex subquery
x. set-inequality query --> compound subquery --> Does-not-contain subquery
xi. compound subquery --> binary subquery

2.3 의미 추출 단계

이 단계는 이름 변환 단계와 전처리 단계를 통하여 생성된 SQL 질의를 관계 대수 연산식으로 변환시킨다. 현재 질의 처리 시스템은 비중첩 질의만을 처리하지만 관계 대수로의 변환은 각 부질의, 단순 질의, group-by 질의, exists 질의, complex

질의, binary 질의를 각각 해당되는 관계 대수 연산식으로 변환시킨다.

(1) 단순 질의

단순 질의는 다음과 같은 형태로 변환된다.

PJ[Q.<selector>] SL[Q.<simple-predicate>] car(Q.<relation-list>)

Simple query를 관계 대수로 변환시키는 알고리즘은 다음과 같다.

```
meaning(Q: simple query) =
if empty(Q.<simple_predicate>)
then
    PJ[Q.<selector>]
    FN[Q.<selector>.<function_list>; {} ]
    car(Q.<relation_list>)
else
    PJ[Q.<selector> U other(Q)]
    FN[Q.<selector>.<function_list>;other(Q)]
    SL[Q.<simple_predicate>]
    car(Q.<relation_list> U extrels(Q.<simple_predicate>))

car(set of relational expressions) = 각 관계 대수 식의 cartesian product
other(Q) = attr(extrels(Q.<simple_predicate>) -
  Q.<relation_list>)
attr(set of relations) = 릴레이션들에 속하는 속성의 집합
```

예.

```
SELECT R.A
FROM R

=> PJ[R.A]R
```

(2) group-by 질의

group-by 질의는 HAVING절의 유무와 통계 함수의 유무에 따라 네가지 경우로 구별하여 각각의 경우에 해당되는 관계 대수 연산으로 변환한다.

```
meaning(Q : group-by query) =
    if empty(Q.<hav_condition>)
    then if empty(Q.<selector>.<function_spec_list>)
  then meaning(Q.<unary_query>
  else
        PJ[Q.<unary_query>.<selector>]
        FN[Q.<unary_query>.<selector>.<function_spec_list>; Q.<gb_attr>]
        meaning(Q.<unary_query>)
  else
        if Q.<hav_condition>.3 = <constant>
        then
   PJ[Q.<unary_query>.<selector>]
   FN[Q.<unary_query>.<selector>.<function_spec_list> -
        Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
   SL[Q.<hav_condition>]
   FN[Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
   meaning(Q.<unary_quer>)
        else
   PJ[Q.<unary_query>.<selector>]
   FN[Q.<unary_query>.<selector>.<function_spec_list -
        Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
   (FN[Q.<hav_condition>.<function_spec>; Q.<gb_attr>]
    meaning(Q.<unary_query>)
   SJ[Q.<hav_condition>.<function_spec>
        Q.<hav_condition>.<comp_op>
        Q.<hav_condition>.<ngb_query>.<selector>]
   meaning(Q.<hav_condition>.<ngb_query>)
```

예.

```
SELECT F1(R.A)
FROM R
WHERE R.C=7
GROUP BY R.B
HAVING F2(R.C) > 2

==> PJ[F1(R.A)] FN[F1(R.A),B] SL[F2(R.C) > 2] FN[F2(R.C),B]
      meaning(SELECT R.A, R.B, R.C FROM R WHERE R.C = 7)
```

(3) exists 질의

exists 질의는 질의 내에 group-by 질의의 유무에 따라 해당되는 관계 연산으로 변환한다.

```
meaning(Q: exists query) =
PJ[Q.<selector> U other(Q)]
FN[Q.<selector>.<function_spec_list>; other(Q)]
car((Q.<relation_list> - connect(Q)) U meaning(Q.<ngb_query>))

connect(Q) = r(meaning(Q.<ngb_query>)) -
        (Q.<ngb_query>.<relation_list>)
other(Q) = attr(connect(Q) - Q.<relation_list>)
r(relational expression) = 관계 식에서 나타나는 릴레이션의 집합
```

예.

```
SELECT R.A
FROM R
WHERE EXISTS SELECT S.A FROM S WHERE S.B = R.A

=> PJ[R.A](Q.<PJ[S.A,R.A,R.B]SL[S.B = R.A](S CP R)]>)
```

(4) complex 질의

complex 질의는 비교 연산을 selection 연산으로 변환시켜 중첩된 관계 대수 연산으로 변환한다.

```
meaning(Q: complex query) =
    PJ[Q.<selector>.<function_spec_list>; other(Q)]
    SL[Q.<left_term>
        Q.<comp_op>
        Q.<ngb_query>.<selector>]
    car((Q.<relation_list> connect(Q)) U
          meaning(Q.<ngb_query>))

    connect(Q) = r(meaning(Q.<ngb_query>)) -
         (Q.<ngb_query>.<relation_list>)
    other(Q) = attr(connect(Q) - Q.<relation_list>)
```

예.

```
SELECT S.A
FROM S
WHERE S.C = SELECT R.C FROM R WHERE R.B=S.B

=>PJ[S.A]SL[S.C = R.C]PJ[R.C,S.A,S.B,S.C]SL[R.B=S.B](R CP S)
```

(5) binary 질의

binary 질의는 INTERSECT, UNION, DIFFERENCE를 해당되는 관계 대수 연산 IN, UN, DF로 변환시킨다.

```
meaning(Q: binary query) =
    meaning(Q.<ngb_query_1>) tr(Q.<set_op>)meaning(Q.<ngb_query_2>)

    with tr(INTERSECT) = IN[other1(Q); other2(Q)]
         tr(UNION)      = UN[other1(Q); other2(Q)]
         tr(MINUS)      = DF[other1(Q); other2(Q)]

    other1(Q) = a(meaning(Q.<ngb_quer_1>)) -
                attr(Q.<ngb_query_1>.<relation_list>)

    other2(Q) = a(meaning(Q.<ngb_query_2>)) -
         attr(Q.<ngb_query_2>.<relation_list>)
```

예.

```
SELECT R.A FROM R
WHERE EXISTS
   (SELECT S.B
    FROM S
    INTERSECT SELECT T.B FROM T WHERE T.C = R.C)

==>PJ[R.A]((PJ[S.B] S CP R) IN
    (PJ[T.B,R.A,R.C] SL [T.C = R.C] (T CP R)))
```

3.4 후처리 단계

후처리 단계는 의미 추출 과정에서 생성된 관계 대수 연산에 적용된다.

(1) 릴레이션의 cartesian product에 행해진 selection은 조인으로 변환시킨다.

예.

```
SL[R.A = S.A]SL[S.A>7](R CP S)

=> R JN[R.A = S.A]SL[S.A>7]S
```

(2) 수행 결과에 영향을 미치지 않는 프로젝션을 찾아 관계 대수 연산에서 제거한다.

예.

PJ[R.A]PF[R.A, R.B]PJ[R.A,R.B,R.C]SL[R.D>8]R

=> PJ[R.A]SL[R.D>8]R

(3) 임시 릴레이션과 관계되는 연산식들을 하나의 연산식으로 구성한다.

관계 대수 변환 프로그램을 통하여 구성된 관계 대수의 내부구조는 그림 3.3과 같다. 그림 3.3에서 algebra 구조의 rel_op는 선택, 프로젝션, 조인 등의 연산을 나타내며 method, lidxid, ridxid, key, lb, ub의 값은 질의 수행 시에 필요한 정보로서 질의 최적화 프로그램에서 결정한다. rel_name은 릴레이션의 이름이다. 관계 대수 구조는 트리 구조이므로 left, right은 하나의 관계 대수 노드의 자 노드(child node)에 대한 포인터이고, parent는 부 노드(parent node)에 대한 포인터이다. pj_list는 관계 대수 노드가 프로젝션 노드인 경우 프로젝션 리스트의 구조를 가리키며 predicate는 선택 또는 조인 노드인 경우 그 연산의 프레디킷트 구조이다. 프레디킷트 구조도 트리 구조이며 또한 b_primary는 하나의 관계식(예. A > B)을 나타내고 이러한 b_primary들의 AND, OR, NOT으로 프레디킷트 구조는 구성된다. func_spec_list는 관계 대수 노드가 통계 함수를 나타내는경우에 구해야 하는 함수의 종류로 구성된다.

제 3 절 최적화 방법

본 장에서는 질의 처리 프로그램 내의 최적화 방법의 설계와 구현에 대해 기술한다. 본 질의 처리 시스템은 System R의 질의 최적화 방법[Seli 79]을 기본으로 하여 다음 세가지 면에서 성능을 향상시키는 것을 목적으로 한다:

(1) 연속 블럭(extent)의 구현

(2) 조인 방법에 대한 휴리스틱(heuristic) 사용

(3) 빠른 죠인 방법 사용.

(1) 연속 블럭(extent)의 구현: 연구용 INGRES와 Troll은 모두 UNIX 화일 시스템을 기반으로 하고 있기 때문에 물리적으로 연속된 블럭을 한 화일에 배당할 수 없다. System R에서는 페이지들의 논리적 단위로 세그먼트라는 개념으로 연속 블럭을 구현하였다. 상업용 INGRES도 연속 블럭(extent)이라는 개념으로 물리적으로 연속된 블럭을 제공하고 있으므로 순차 처리를 빠르게 실행할 수 있다.

(2) 죠인 방법에 대한 휴리스틱 사용: System R의 탐색 트리에서는 가능한 모든 죠인 방법에 대해 탐색 트리를 구성한다. 본 시스템에서는 각 릴레이션에 대한 탐색 트리가 완성된 시점에서 이 탐색 트리와 릴레이션에 대한 정보(예를 들어 크기, 튜플 수)를 고려하여 최적에 가까운 죠인 방법을 정한다. 예를 들어 어떤 두 릴레이션 사이의 죠인에 대해 두 릴레이션이 모두 죠인 속성(join attribute)에 대해 다발 색인(clustered index)을 갖고 있으면 병합 스캔(merging scan)이 두 릴레이션의 각 페이지를 한번씩만 액세스하면 되므로 성능이 가장 우수한 방법이다.

(3) 빠른 죠인 방법 이용: 앞에서 언급했듯이 죠인 실행을 위해 INGRES에서는 중포 루프(nested loop) 방법, System R에서는 중포 루프 방법과 병합 스캔(merging scan) 방법을 구현 하고 있다. 본 연구에서는 중포 루프(nested loop)와 병합 스캔(merging scan), 블럭 중포(block nested) 등의 세 가지 죠인 방법을 구현 했으며, 각 죠인 연산에 대해 가장 우수한 죠인 방법을 선택하여 실행함으로써 성능 향상을 도모하였다.

본 연구에서 구현된 질의 최적화 프로그램은 크게 관계 대수 최적화 단계와 액

세스 방법 최적화의 두 단계로 나누어진다. 관계 대수 최적화 단계는 System R의 최적화 방법과 유사하며 select-down과 projection-down 등을 실행한다. Select-down은 검색할 튜플에 대한 조건을 가능하면 먼저 비교 하도록 관계 대수 트리에서 조건을 밑으로 내린다. 그리고 projection-down에서는 검색할 튜플의 애트리뷰트를 선택하는 프로젝션 리스트를 가능하면 먼저 만들도록 관계 대수 트리에서 프로젝션 리스트를 밑으로 내린다. 예를들어 다음과 같은 데이타베이스 스키마가 정의 되어 있을때

```
Books(Title, Author, Pname, LC_NO)
Publishers(Pname, Paddr, Pcity)
Borrowers(Name, Addr, City, Card_NO)
Loans(Card_NO, LC_NO, Date)
```

다음과 같은 질의를 파싱해서 나온 관계 대수 트리는 그림 3.4와 같다.

```
SELECT    Title, Author, Pname, LC_NO,
          Name, Addr, City, Card_NO, Date
FROM      Loans, Borrowers, Books
WHERE     Borrowers.Card_NO = Loans.Card_NO
AND       Books.LC_NO = Loans.LC_NO
```

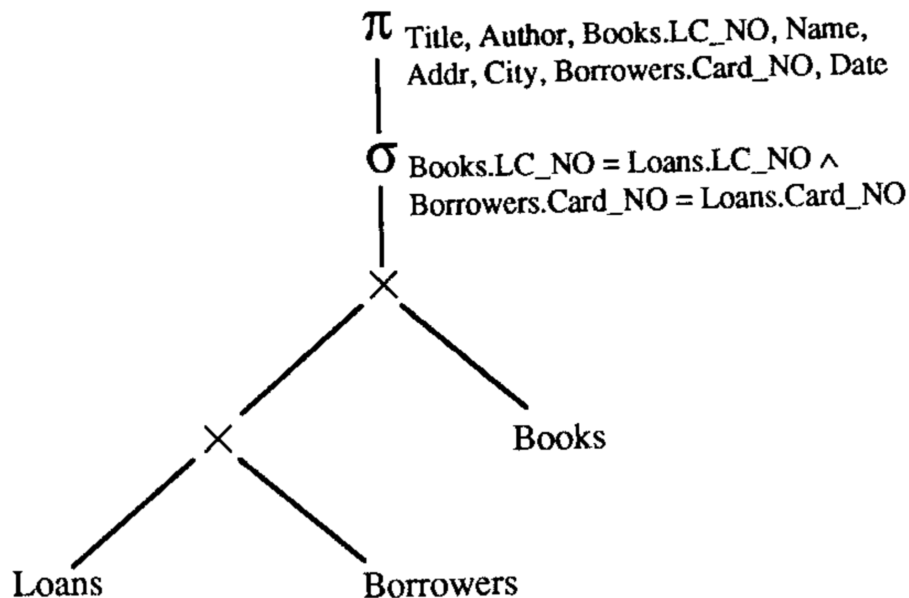


그림 3.4 관계 대수 트리

P = Title, Author, Pname, LC_NO, Name, Addr, City, Card_NO, Date 이고
F = Borrowers.Card_NO = Loans.Card_NO and Books.LC_NO = Loans.LC_NO
라고 할때, 앞의 질의는 $\pi_P(\sigma_F(\text{Loans} \times \text{Borrowers} \times \text{Books})$ 와 같은 관계 대수 식으로 나타낼 수 있다. 그림 3.1의 관계 대수 트리를 select-down과 projection-down을 해서 최적화 하면 그림 3.5와 같다.

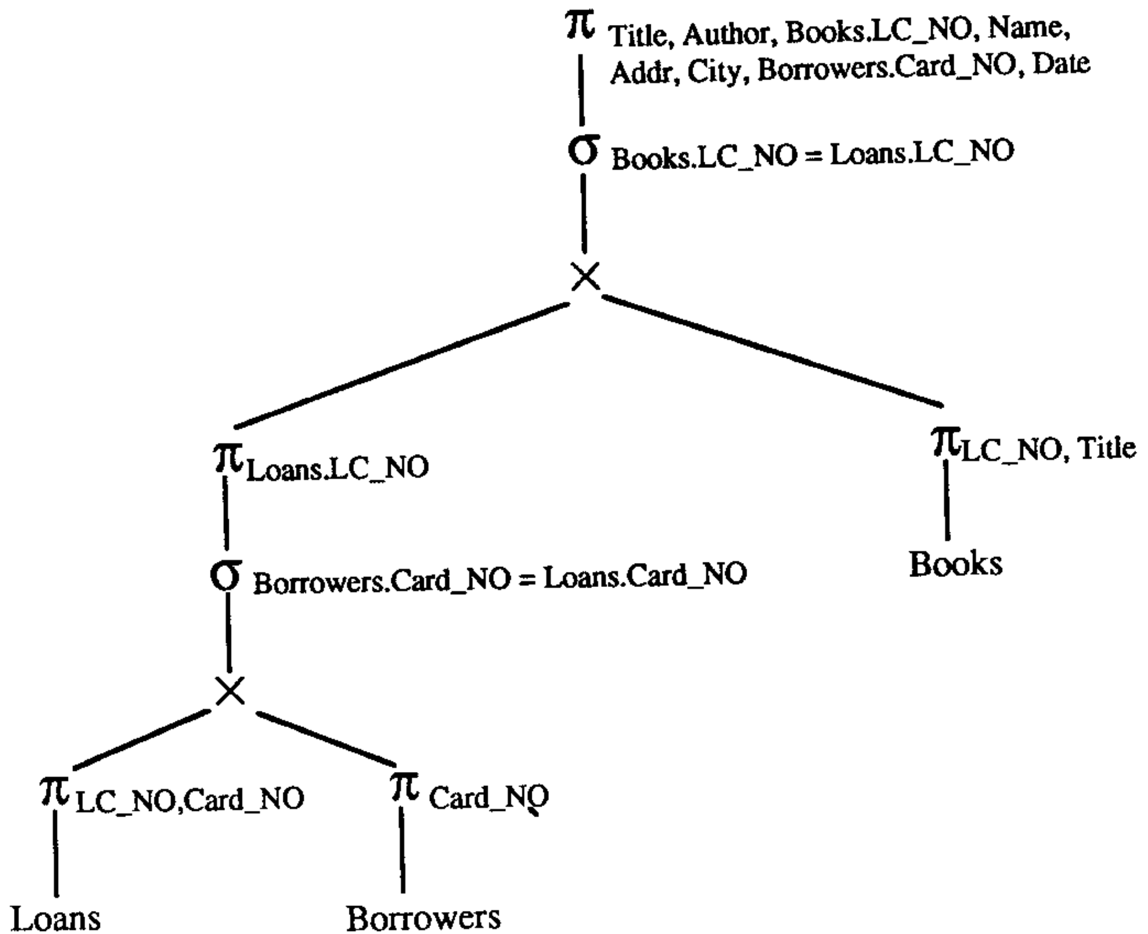


그림 3.5 최적화된 관계 대수 트리

액세스 방법 선택 단계에서는 조건을 만족하는 튜플을 검색할 때 가장 빠른 시간에 검색할 수 있도록 하는 액세스 방법에 대한 최적화이다. System R에서는 비용 함수(Cost Function)를 써서 비용이 가장 적은 액세스 방법을 선택한다. 그러나 비용 함수에 의해서 계산한 실행비용이 실제 실행 시간을 잘 예측하지 못하는 경우가 많다[Loth 86]. 또 한 질의를 실행하는 방법이 지수 함수적으로 늘어남으로 각 방법의 모든 실행 비용을 예측하는데 상당한 부가 노력이 소요된다.본 연구에서는 이와 같은 문제점을 해결하기 위해 휴리스틱 방법[Meec 87]을 이용한다. 이 방법은 각각의 형태의 질의를 데이타베이스의 여러 상황에 대해 실제 실행해서 비슷한 상황에서 비슷한 형태의 질의가 제기되면 그 방법을 택한다.

두개 이상의 릴레이션을 가지고 각 릴레이션의 애트리뷰트를 비교하는 죠인 질의에 대해서는 본 연구에서 구현된 3가지 죠인 방법인 중포 루프(nested loop), 병합 스캔(merging scan), 블럭 중포(block nested)에 대해서 그중 하나를 선택 하도록 한다.

중포 루프 죠인 방법에서는 죠인할 테이블을 버퍼로 읽어 들이고 이 테이블과 죠인될 테이블의 인덱스를 통해서 검색한다. 병합 스캔 죠인 방법에서는 죠인할 두 테이블을 버퍼로 읽어들이고 두 테입블의 튜플을 스캔하면서 색한다. 블럭 중포 죠인 방법에서는 한 테이블의 모든 튜플을 버퍼로 읽어서 나머지 한 테이블을 검색한다.

죠인 방법을 선택하기 위해서 죠인 테이블의 인덱스 유무와 테이블의 크기를 고려한다. 두 죠인 테이블 중 하나의 테이블에만 죠인 애트리뷰트에 대한 인덱스가 있고 테이블의 크기가 큰 경우 중포 루프 방법을 선택한다. 두 죠인 테이블이 죠인 애트리뷰트에 대해서 정렬이 되어 있는 경우 병합 스캔 방법이 선택되고 죠인 애트리뷰트에 대해서 인덱스도 없고 정렬되어 있지 않은 경우에 블럭 중포 방법을 선택한다.

구현된 질의 최적화 프로그램에 대해서 [Dina 84]의 벤치마크 데이타베이스에 대해서 실험해 보았다. 벤치마크 데이타베이스는 4개의 테이블 thoustup, twothoustup, fivethoustup, tenthoustup 로 구성되어 있으며 각 테이블의 애트리뷰트는 아래 그림 3.6과 같이 unique1, unique2, two, ten, hundred, thoustup 애트리뷰트로 구성되어 있으며, 각 애트리뷰트는 4 바이트의 정수 이다. 각 테이블의 키는 unique1과 unique2 이고 각 테이블의 인덱스로 사용된다.

unique1	unique2	two	ten	hundred	thousand
285	0	0	5	84	225
582	1	1	6	77	770
19	2	0	1	22	311
316	3	1	2	23	784
17	4	0	3	48	605
570	5	1	8	5	878
479	6	0	5	30	91
680	7	1	0	19	260
477	8	0	9	88	129
318	9	1	4	49	266
739	10	0	7	18	895
340	11	1	4	11	120
609	12	0	1	4	909
930	13	1	4	1	814
639	14	0	7	6	163
640	15	1	2	83	60
581	16	0	1	4	769
158	17	1	6	61	322
67	18	0	5	58	815
764	19	1	0	75	384
9	20	0	9	76	981
290	21	1	6	97	710
103	22	0	1	98	539
488	23	1	0	99	172
130	24	1	2	81	126
191	25	0	1	90	747
664	26	1	6	19	756

그림 3.6 thoustup 릴레이션의 일부분

이 벤치마크 데이타베이스에 대해서 실험을 한 결과 두 조인 애트리뷰트에 대한 인덱스가 존재할 때 중포 루프와 병합 스캔 그리고 블럭 중포 조인 방법에 대해서 병합 스캔 방법이 가장 좋았고, 릴레이션의 크기가 작을때에는 인덱스의 존재와는 상관없이 블럭 중포 방법이 좋은 성능을 나타냈다.

제 4 절 관계 대수 연산자의 구현

본 시스템이 제공하는 질의어를 처리하기 위해서는 선택(selection)과 프로젝션(projection), 조인(join) 등의 세 관계 대수 연산이 필요하다. 이들 각 연산을 수행하기 위해 각 연산에 대해 성능이 우수한 방법들을 구현하여 성능을 비교한다.

본 연구의 질의 시스템은 중포 루프(nested loop) 방법과 병합 스캔(merging scan) [Seli 79], 블럭 중포(block nested) 등의 세 가지 조인 방법을 구현하였다. 중포 루프 방법은 조인 선택도가 낮은 경우나 한 릴레이션의 크기가 아주 작은 경우에 우수한 성능을 제공한다[Blas 77, Shap 86]. 병합 스캔 방법[Blas 77]은 조인에 참가하는 두 릴레이션이 조인 속성(join attribute)에 대해 소팅(sorting)되어 있거나, 다발 색인(clustered index)을 갖고 있을 경우에 가장 우수한 성능을 제공한다. 블럭 중포 방법은 조인에 참가하는 두 릴레이션의 크기가 작을때 좋은 성능을 제공한다.

3.4.1 선택과 프로젝션의 구현

선택에 대해서는 새로운 방법을 설계하지 않고 현재 저장 시스템에서 액세스 방법으로 제공하고 있는 순차 스캔(sequential scan)과 B^+-트리나 해쉬 색인을 통한 색인 스캔(index scan)을 이용한 방법을 구현한다. 프러젝션은 전체 튜플을 스캐하면서 해당되는 속성만 출력하면 되므로 본 시스템에서는 선택과 프러젝션 사이에 임시 릴레이션을 만드는 부가 노력을 덜기 위해 다음과 같이 한 모듈로 선택과 프러젝션

을 구현한다.

```
int ex_select(DbId, IRel, ORel, PAttr, Expr, Key, Lb, Ub, Idxid)
int             DbId;
char            *IRel;
char            *ORel;
FIELDLIST       PAttr;
BOOLEXP         Expr;
IDXKEYINFO      Key;
IDXKEY          Lb;
IDXKEY          *Ub;
int             dxid;
```

데이타베이스 DbId의 릴레이션 IRel에서 Expr을 만족하는 튜플들을 선택하여 PAttr에 나타나는 속성으로 프로젝션하여 ORel을 통해 돌려준다. PAttr이 NULL이면 선택만 실행하고 Expr이 NULL이면 선택만 실행하고 ORel이 NULL이면 결과를 화면으로 출력한다. 이 과정에서 사용하는 스캔 방법은 선택 조건의 애트리뷰트에 대한 인덱스가 있으면 인덱스를 통해서 하고, 없으면 순차 스캔을 한다. 돌려주는 값이 양수 일때는 선택된 튜플의 수이고 음수 일때에는 에러가 발생한 경우이다.

```
int ex_method(DbId, Rel, Expr, RExpr, Key, Lb, Ub)
int             DbId;
char            *Rel;
BOOLEXP         *Expr;
BOOLEXP         **RExpr;
IDXKEYINFO      Key;
IDXKEY          *Lb;
IDXKEY          *Ub;
```

데이타베이스 DbId의 릴레이션 Rel에서 Expr을 만족하는 튜플을 찾는 최적의 스캔 방법을 구한다. 기본적인 방법은 다음과 같다.

1. Relation이 차지하는 페이지 수가 정해진 값 이하이거나 이 릴레이션이 색인을 갖고 있지 않으면 순차 스캔을 택한다.
2. Expr이 "OR"나 "LIKE", "NE" 등의 연산자를 포함하고 있으면 순차 스캔을 택한다.

3. Expr이 다발형 색인에 대해 색인 사용이 가능한(sargable) 프레디킷트를 갖고 잇으면 이 색인을 이용한 스캔을 택한다.
4. Expr이 비다발형 색인에 대해 색인 사용이 가능한(sargable) 프레디킷트를 갖고 있으면 이 색인을 이용한 스캔을 택한다.
5. 위의 어느 경우에도 해당되지 않으면 순차 스캔을 택한다.

함수값이 0인 경우는 순차 스캔을 나타내며, 1 이상인 경우에는 그 함수값이 나타내는 색인을 사용한 스캔을 의미한다. 색인을 사용한 스캔을 택한 경우 Key, Lb, Ub를 통해 그 색인을 이용해 찾을 키 속성과 그 최저값, 최고값을 돌려준다. 또 RExpr을 통해 색인 스캔의 최저값과 최고값에 의해 제거할 수 있는 프레디킷트를 제거한 후의 Expr을 돌려준다.

3.4.2 중포 루프 죠인 방법의 구현

중첩 루프 방법은 가장 쉽게 구현할 수 있는 죠인 방법이며, 양 릴레이션을 순차 스캔을 통해 액세스한다. 이 방법은 죠인의 선택도가 낮을 때 좋은 성능을 제공한다.

```
int ex_nestjoin(DbId, Outer, Inner, Return, RAttr, PList, Expr, Iidx, Clist)
int             DbId;
int             Iidx;
char            *Inner;
char            *Outer;
char            *Return;
JOINLIST        RAttr;
PREDLIST        PList;
BOOLEXP         *Expr;
CHGLIST         **Clist;
```

데이타베이스 DbId에서 릴레이션 Inner와 Outer를 순차 스캔을 통한 중첩 루프 방식으로 죠인한 후 RAttr로 프로젝션하여 릴레이션 Return을 통해 결과를 돌려준다. 스캔 방법은 ex_method를 사용하여 구한다. 돌려주는 값이 양수 일때는 죠인된 튜플

의 수 이고 음수 일때에는 에러가 발생한 경우이다.

3.4.3 병합 스캔 방법의 구현

이 방법은 양 릴레이션이 조인하는 속성에 대해 트리 색인을 갖고 있을 경우, 특히 다발 색인을 갖고 있을 경우에, 우수한 성능을 제공한다. 해쉬 색인은 순서화된 스캔을 제공하지 못하므로 병합 스캔에 사용할 수 없다. 이 방법은 양 릴레이션을 조인 속성의 순서대로 스캔하면서 병합하는 방법이므로 동호-조인(equi-join)에만 사용한다. 조인의 선택도가 높을 때 우수한 성능을 제공한다.

```
int ex_mergejoin(DbId, Outer, Inner, Return, RAttr, PList,
        OExpr, IExpr, Oidx, Iidx, Clist)
int             DbId;
int             Oidx;
int             Iidx;
char            *Inner;
char            *Outer;
char            *Return;
JOINLIST        RAttr;
PREDLIST        PList;
BOOLEXP         *OExpr;
BOOLEXP         *IExpr;
CHGLIST         **Clist;
```

데이타베이스 DbId에서 릴레이션 Rel1과 Rel2를 트리 색인을 이용한 스캔을 통해 병합 조인한 후, RAttr로 프로젝션하여 릴레이션 Return을 통해 결과를 돌려준다. 돌려주는 값이 양수 일때는 조인된 튜플의 수 이고 음수 일때에는 에러가 발생한 경우이다.

3.4.3 블럭 중포 방법의 구현

이 방법은 조인에 참가하는 릴레이션 중 한 릴레이션의 크기가 작을 경우 우수한 성능을 제공한다.

```
int ex_blockjoin(DbId, Outer, Inner, Return, RAttr, PList, Expr, Clist)
int              DbId;
char             *Inner;
char             *Outer;
char             *Return;
JOINLIST         RAttr;
PREDLIST         PList;
BOOLEXP          *Expr;
CHGLIST          **Clist;
```

데이타베이스 DbId에서 릴레이션 Inner를 버퍼로 읽어들이고 릴레이Outter를 스캔해 나가면서 PList의 조건에 맞는 튜플들을 선택해서 릴레이션 Return을 통해서 결과를 돌려준다. 돌려주는 값이 양수 일때는 조인된 튜플의 수 이고 음수 일때에는 에러가 발생한 경우이다.

3.4.4 각종 함수의 구현

본 연구에서 구현한 함수는 최대(max), 최소(min), 평균(avg), 합(sum), 개수(count) 이다.

```
int ex_fn(DbId, IRel, ORel, GAttr, Fnlist)
int              DbId;
char             *IRel;
char             *ORel;
FIELDLIST        GAttr;
FNLIST           *Fnlist;
```

데이타베이스 DbId에서 릴레이션 Irel의 함수 리스트 Fnlist에 있는 함수들의 그룹 애트리뷰트 리스트 Gattr에 대해서 계산한 결과를 릴레이션 Orel로 돌려준다. 함수를 계산할때 해당 함수에 대해서 각각 ex_max, ex_min, ex_avg, ex_sum, ex_cnt를 호출한다. 돌려주는 값이 양수 일때는 선택된 튜플의 수 이고 음수 일때에는 에러가 발생한 경우이다.

```
int ex_max(scan, size, match, attr, MaxVal)
int              scan;
```

```
int                 size;
int                 match;
FNLIST              attr;
char                *MaxVal;
```

이 함수는 ex_fn에서 함수가 max일때 호출되고 결과를 MaxVal에 돌려준다. 돌려주는 값이 양수 일때는 최대값을 계산하기 위해 참여한 튜플의 수 이고 음수 일때에는 에러가 발생한 경우이다.

```
int ex_min(scan, size, match, attr, MinVal)
int                 scan;
int                 size;
int                 match;
FNLIST              attr;
char                *MinVal;
```

이 함수는 ex_fn에서 함수가 min일때 호출되고 결과를 MinVal에 돌려준다. 돌려주는 값이 양수 일때는 최소값을 계산하기 위해 참여한 튜플의 수 이고 음수 일때에는 에러가 발생한 경우이다.

```
int ex_avg(scan, size, attr, AvgVal)
int                 scan;
int                 size;
FNLIST              attr;
char                AvgVal;
```

이 함수은 ex_fn에서 함수가 avg일때 호출되고 결과를 AvgVal에 돌려준다. 돌려주는 값이 양수 일때는 평균이 계산된 튜플의 수 이고 음수 일때에는 에러가 발생한 경우이다.

```
int ex_sum(scan, size, attr, SumVal)
int                 scan;
int                 size;
FNLIST              *attr;
char                *SumVal;
```

이 함수는 ex_fn에서 함수가 sum일때 호출되고 결과를 SumVal에 돌려준다. 돌려주는 값이 양수 일때는 합이 계산된 튜플의 수 이고 음수 일때에는 에러가 발생한

경우이다.

```
int  ex_cnt(scan,  size,  Ntuple)
int                 scan;
int                 size;
char                *Ntuple;
```

이 함수는 ex_fn에서 함수가 count일때 호출되고 결과를 Ntuple에 돌려준다. 돌려주는 값이 양수 일때는 선택된 튜플의 수 이고 음수 일때에는 에러가 발생한 경우이다.

제 5 절 질의를 제외한 데이타 조작어 구현

이 절에서는 한 릴레이션에 대한 삽입과 삭제, 변경 등의 연산을 구현한다.

```
int  ex_insert(DbId,  Rel,  Value)
int                 DbId;
char                *Rel;
VALUELIST           *Value;
```

데이타베이스 DbId의 릴레이션 Rel에 Value가 표현하는 튜플을 삽입한다. 돌려주는 값이 0이면 에러 없이 정상적으로 수행된 경우이고 음수 이면 에러가 발생한 경우이다.

```
int  ex_delete(DbId,  Rel,  Expr,  Key,  Lb,  Ub,  Idxid)
int                 DbId;
char                *Rel;
BOOLEXP             *Expr;
IDXKEYINFO          Key;
IDXKEY              *Lb;
IDXKEY              *Ub;
int                 Idxid;
```

데이타베이스 DbId의 릴레이션 Rel에서 Expr을 만족하는 튜플들을 제거한다. 인덱스가 존재하는 경우 해당 인덱스도 제거한다. 제거된 튜플의 수를 함수값을 통해

돌려준다. 돌려주는 값이 음수인 경우는 에러가 발생한 경우이다.

```
int ex_update(DbId, Rel, UAttr, Exptree, Expr, Key, Lb, Ub, Idxid, Upd_Exp)
int             DbId;
int             Idxid;
char            *Rel;
FIELDLIST       UAttr;
EXPTREE         *Exptree[];
BOOLEXP         *Expr;
IDXKEYINFO      Key;
IDXKEY          *Lb;
IDXKEY          *Ub;
BOOLEXP         *Upd_Exp;
```

데이타베이스 DbId의 릴레이션 Rel에서 Expr을 만족하는 튜플들을 읽는다. 조건에 맞는 튜플들을 버퍼로 읽어들인 후, UAttr 리스트의 각 속성에 대해 Upd_Exp의 해당되는 식을 실행한 값으로 버퍼 내의 튜플을 대치한다. 변경된 튜플의 수를 함수값을 통해 돌려준다. 돌려주는 값이 음수이면 에러가 발생한 경우이다.

제 6 절 캐털로그 관리 구현

이 장에서는 캐털로그 시스템의 관리와 구현에 관하여 기술한다 사용자들이 데이타 정의어를 통해 정의한 데이타베이스 스키마도 데이타와 마찬가지로 릴레이션에 저장되는데 이러한 릴레이션을 사용자 릴레이션과 구별하여 캐털로그 (catalog) 릴레이션이라고 한다. 그림 3.7는 캐털로그 릴레이션들의 구조이다.

SYSDB는 시스템 내에 존재하는 데이타베이스들에 관한 정보로 구성된 캐털로그 릴레이션이다. SYSREL은 사용자 릴레이션에 관한 정보로 구성되며 SYSATT는 사용자 릴레이션을 구성하는 속성에 관한 정보로 구성되어 있으며 SYSIDX는 데이타베이스에 존재하는 색인에 관한 정보로 구성된 캐털로그 릴레이션이다. 이들 캐털로그 릴레이션은 모든 사용자 데이타베이스에 존재하며 질의 최적화 프로그램에서 최적화를

위한 통계 정보를 위해 이들 릴레이션을 액세스하고 질의 수행 프로그램에서 액세스 길을 결정할 때도 캐털로그 릴레이션의 정보를 액세스한다.

SYSDB

dbname	path	dbid	ownerid	nextent	npage

SYSREL

rel-name	owner-name	owner-id	nattr	n-tuple	m-width	npage	page-fill	extent-fill	ctime	utime

SYSATT

attname	relname	attid	offset	length	data_type	nunique

SYSIDX

idx-name	rel-name	idx-id	owner-id	n-attr	attid1	attid2	attid3	attid4	n-level	n-page	ctime	utime	idx-type	unique	clus-ter

그림 3.7 캐털로그 릴레이션의 구조

```
int ca_createdb(Dbname, Path, Nextent, Npage)
char                *Dbname;
char                *Path;
int                 Nextent;
int                 Npage;
```

Dbname이라는 이름의 데이타베이스를 만든다. 새로운 데이타베이스가 생성되는 곳을 Path 정보로 나타낸다. 즉, Path directory에 데이타베이스가 생긴다. 따라서 database 캐털로그 릴레이션인 SYSDB에 새로운 튜플을 추가한다. Nextent는 예상되

는 연속 블록 갯수이고 Npage는 한 연속 블록당 페이지의 갯수이다 (한 페이지의 크기는 4K 바이트이다). Nextent * Npage가 전체 데이타베이스에 할당된 블록의 수이다. 함수값으로 데이타베이스 번호를 돌려주고 만일 데이타베이스를 생성할 수 없는 경우가 생기면 오류 번호를 돌려 준다.

```
int  ca_destroydb(Dbname)
char                  *Dbname;
```

Dbname라는 이름의 데이타베이스를 제거한다. 데이타베이스의 캐털로그 릴레이션 SYSDB에서 Dbname 데이타베이스의 정보에 해당하는 튜플을 삭제하고 그 데이타베이스에 속한 모든 릴레이션, 속성, 색인에 관한 정보를 각각 SYSREL, SYSATT, SYSIDX 릴레이션에서 삭제한다. 만일 Dbname 데이타베이스를 제거할 수 없는 경우이면 오류 번호를 돌려준다.

```
int  ca_createrel(DbId, Relname, attptr, Npage, Pagefill, Extentfill)
int                   DbId;
char                  *Relname;
ATTLIST               *attptr;
int                   Npage;
int                   Pagefill;
int                   Extentfill;
```

데이타베이스 DbId내에 attptr 리스트에 나타난 형태로 릴레이션 Relname을 생성한다. NPage는 이 릴레이션이 사용할 페이지 수의 예상치이고 Pagefill은 페이지의 fill factor이고 Extentfill은 extent의 fill factor이다. 생성하는 릴레이션의 정보를 캐털로그 릴레이션 SYSREL에 수록하고 Relname 릴레이션에 속한 속성에 관한 정보를 SYSATT 릴레이션에 삽입한다. 만일 Relname 릴레이션을 생성할 수 없으면 오류 번호를 돌려준다.

```
int  ca_destroyrel(DbId, Relname)
int                   DbId;
char                  *Relname;
```

데이타베이스 DbId내에서 릴레이션 Relname을 제거한다. 릴레이션 Relname이 차지하고 있던 페이지를 모두 반납하고, 이 릴레이션에 정의된 모든 색인들을 제거한다. 캐털로그 릴레이션 SYSREL에서 Relname 릴레이션에 관한 정보를 삭제하고 그 릴레이션에 속한 속성에 관한 정보를 SYSATT 릴레이션에서 삭제한다. 마찬가지로 Relname 릴레이션에 정의되어 있는 색인이 존재하면 그 색인이 차지하고 있는 페이지를 반납하고 그 색인에 관한 정보를 SYSIDX 릴레이션에서 제거한다. 만일 Relname 릴레이션을 제거할 수 없으면 오류 번호를 돌려준다.

```
int ca_createidx(DbId, Relname, Idxname, Key,
                 Idxtype, Unique, Cluster, Fillfactor)
int              DbId;
int              Idxtype;
int              Unique;
int              Cluster;
int              Fillfactor;
char             *Relname;
char             *Idxname;
IDXKEYINFO       *Key;
```

데이타베이스 DbId에서 Relname 릴레이션에 대하여 Key 구조에 있는 그 색인을 구성하는 속성으로 색인을 생성하고 이름을 Idxname으로 한다. Idxtype에 따라 B-tree 색인, 해싱 색인을 구분하고, Unique, Cluster는 그 색인을 구성하는 애트리뷰트가 unique인가, 결집할 것인가를 결정한다. Fillfactor는 생성되는 색인 화일의 fill factor이다. 생성되는 색인에 관한 정보를 캐터로그 릴레이션 SYSIDX에 삽입한다. 만일 색인을 구성할 수 없으면 오류 번호를 돌려준다.

```
int ca_destroyidx(DbId, Idxname)
int              DbId;
char             *Idxname;
```

데이타베이스 DbId에서 Idxname을 이름으로 하는 색인을 제거한다. 즉 색인 화일을 제거하여 사용하고 있던 페이지를 반납하고 그 색인에 관한 정보를 캐털로그 릴레이션 SYSIDX에서 제거한다. 만일 색인을 제거할 수 없으면 오류 번호를 돌려준

다.

```
int ca_fetchaid(DbId, Relname, Attid, aptr)
int                     DbId;
int                     Attid;
char                    *Relname;
SYSATT                  *aptr;
```

데이타베이스 DbId에서 Relname 릴레이션에 Attid를 속성의 번호로 하는 속성에 관한 정보를 캐털로그 릴레이션 SYSATT에서 검색한다. 그 정보의 튜플이 있으면 aptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int ca_fetchaoff(DbId, Relname, Offset, aptr)
int                     DbId;
int                     Offset;
char                    *Relname;
SYSATT                  *aptr;
```

데이타베이스 DbId에서 Relname 릴레이션에 그 릴레이션의 구조에서 Offset 위치에 있는 속성에 관한 정보를 캐털로그 릴레이션 SYSATT에서 검색한다. 그 정보의 튜플이 있으면 aptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int ca_fetchatt(DbId, Relname, Attname, aptr)
int                     DbId;
char                    *Relname;
char                    *Attname;
SYSATT                  *aptr;
```

데이타베이스 DbId에서 Relname 릴레이션에 Attname의 이름을 갖는 속성에 관한 정보를 캐털로그 릴레이션 SYSATT에서 검색한다. 그 정보의 튜플이 있으면 aptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int ca_fetchdb(Dbname, bptr)
char                    *Dbname;
SYSDB                   *bptr;
```

Dbname 이름을 갖는 데이타베이스에 관한 정보를 캐털로그 릴레이션 SYSDB에

서 검색한다. 그 정보의 튜플이 있으면 bptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int ca_fetchdbid(DbId, bptr)
int                DbId;
SYSDB              *bptr;
```

DbId를 번호로 갖는 데이타베이스에 관한 정보를 캐털로그 릴레이션 SYSDB에서 검색한다. 그 정보의 튜플이 있으면 bptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int ca_fetchidx(DbId, Idxname, iptr)
int                DbId;
char               *Idxname;
SYSIDX             *iptr;
```

데이타베이스 DbId에서 Idxname의 이름을 갖는 색인에 관한 정보를 캐털로그 릴레이션 SYSATT에서 검색한다. 그 정보의 튜플이 있으면 iptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int ca_fetchrel(DbId, Relname, rptr)
int                DbId;
char               *Relname;
SYSREL             *rptr;
```

데이타베이스 DbId에서 Relname의 이름을 갖는 릴레이션에 관한 정보를 캐털로그 릴레이션 SYSREL에서 검색한다. 그 정보의 튜플이 있으면 rptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int ca_fetchxid(DbId, Relname, Idxid, iptr)
int                DbId;
int                Idxid;
char               *Relname;
SYSIDX             *iptr;
```

데이타베이스 DbId에서 Relname의 릴레이션에 Idxid를 색인 번호로 하는 색인

의 정보를 캐털로그 릴레이션 SYSIDX에서 검색한다. 그 정보의 튜플이 있으면 iptr에 그 정보를 돌려준다. 만일 없으면 오류 번호를 돌려준다.

```
int  ca_firstatt(DbId,  scan,  aptr,  bool)
int                      DbId;
int                      *scan;
SYSATT                   *aptr;
BOOLEXP                  *bool;
```

데이타베이스 DbId에서 캐털로그 릴레이션 SYSATT에서 어떤 조건 bool을 만족하는 속성에 관한 정보를 검색한다. 예를 들면 어떤 릴레이션에 속하는 속성을 검색할 수 있다. 속성이 있으면 aptr에 그 정보를 돌려주고 연속하여 bool 조건을 만족하는 다른 속성에 관한 정보를 검색하는 경우를 위하여 scan을 돌려 준다. 만일 만족하는 속성이 없으면 오류 번호를 돌려준다.

```
int  ca_firstidx(Dbid,  scan,  iptr,  bool)
int                      Dbid;
int                      *scan;
SYSIDX                   *iptr;
BOOLEXP                  *bool;
```

데이타베이스 DbId에서 캐털로그 릴레이션 SYSIDX에서 어떤 조건 bool을 만족하는 속성에 관한 정보를 검색한다. 에를 들면 어떤 릴레이션에 속하는 색인을 검색할 수 있다. 색인이 있으면 iptr에 그 정보를 돌려주고 연속하여 bool 조건을 만족하는 다른 색인에 관한 정보를 검색하는 경우를 위하여 scan을 돌려 준다. 만일 만족하는 색인이 없으면 오류 번호를 돌려준다.

```
int  ca_firstrel(DbId,  scan,  rptr)
int                      DbId;
int                      *scan;
SYSREL                   *rptr;
```

데이타베이스 DbId에서 캐털로그 릴레이션 SYSREL에서 첫번째 릴레이션에 관한 정보를 검색한다. 릴레이션이 있으면 rptr에 그 정보를 돌려주고 연속하여 다른 릴레

이션에 관한 정보를 검색하는 경우를 위하여 scan을 돌려 준다. 만일 만족하는 릴레이션이 없으면, 즉 그 데이타베이스에 릴레이션이 없으면 오류 번호를 돌려준다.

```
int ca_idxattr(DbId, Relname, aiptr, iptr, match)
int             DbId;
char            *Relname;
ATTIDLIST       *aiptr;
SYSIDX          *iptr;
int             *match;
```

데이타베이스 DbId에서 캐털로그 릴레이션 SYSIDX에서 aiptr에 있는 속성으로 구성된 색인을 검색한다. 만족하는 색인이 여러개인 경우에는 색인을 구성하는 속성과 비교하여 가장 근접하는 색인을 선택하여 iptr에 그 정보를 돌려주고 match에 그 색인을 구성하는 속성과 일치하는 속성의 수를 돌려준다. 만일 만족하는 색인이 없으면 오류 번호를 돌려준다.

```
int ca_idxclus(DbId, Relname, iptr)
int             DbId;
char            *Relname;
SYSIDX          *iptr;
```

데이타베이스 DbId에서 Relname 릴레이션에 구성되어 있는 색인 중 결집된 (cluster) 색인에 관한 정보를 캐털로그 릴레이션 SYSIDX에서 검색한다. 검색된 색인에 관한 정보를 iptr에 돌려준다. 만일 결집 색인이 없으면 오류 번호를 돌려준다.

```
int ca_nextatt(scan, aptr, fno)
int             scan;
int             fno;
SYSATT          *aptr;
```

ca_firstatt에서 돌려 받은 scan을 사용하여 다음 속성에 관한 정보를 검색한다. 검색하여 얻은 속성에 관한 정보를 aptr에 돌려준다. 만일 만족하는 속성이 없으면, 즉 더 이상의 속성이 없으면 오류 번호를 돌려준다.

```
int ca_nextidx(scan, iptr, fno)
```

```
int             fno;
int             scan;
SYSATT          *iptr;
```

ca_firstidx에서 돌려 받은 scan을 사용하여 다음 색인에 관한 정보를 검색한다. 검색하여 얻은 색인에 관한 정보를 iptr에 돌려준다. 만일 만족하는 색인이 없으면, 즉 더 이상의 색인이 없으면 오류 번호를 돌려준다.

```
int ca_nextrel(scan, rptr, fno)
int             scan;
SYSREL          *rptr;
int             fno;
```

ca_firstrel에서 돌려 받은 scan을 사용하여 다음 릴레이션에 관한 정보를 검색한다. 검색하여 얻은 릴레이션에 관한 정보를 rptr에 돌려준다. 만일 만족하는 릴레이션이 없으면, 즉 더 이상의 릴레이션이 없으면 오류 번호를 돌려준다.

제 4 장 트랜잭션 스케줄러

본 장에서는 트랜잭션 스케쥴러 프로그램의 설계와 구현에 대해 기술한다. 1절에서는 트랜잭션과 동시성 제어, 교착상태등의 기본 개념에 대해 기술한다. 2절에서는 본 연구에서 동시성 제어 기법으로 채택한 술어 로킹에 대해 기술하고 3절에서는 술어들간의 상충성 검사 알고리즘에 대해 설명한다. 마지막으로 4절에서는 트랜잭션 스케쥴러 프로그램의 구현에 대해 상세히 기술한다.

제 1 절 기본 개념

4.1.1 트랜잭션

트랜잭션은 데이타베이스에 대한 연산들을 포함하는 응용의 논리적인 작업 단위이다. 환언하면, 트랜잭션은 사용자 프로그램중 원자적(atomic)으로 실행되어야 하는 부분이다. 따라서, 트랜잭션은 부분적으로 실행되어서는 안되고 반드시 완전하게 실행되든지 또는 전혀 실행되지 않아야 한다. 이러한 성질때문에 트랜잭션 개념은 많은 데이타 처리 응용에서 동시 처리와 고장에 대응하여야 하는 사용자의 부담을 덜어주는 데에 이용되어 왔다. 이러한 점에서, 트랜잭션은 일관성(consistency)과 회복(recovery)을 유지하는 단위로 이용된다.

트랜잭션은 정상적으로 혹은 비정상적으로 끝날 수 있다. 트랜잭션이 정상적으로 끝나고 실행된 효과가 영구히 보존되는 경우에 트랜잭션은 완료(commit)되었다고 한다. 그렇지 않은 경우에 트랜잭션은 철회(abort)되었다고 한다. 트랜잭션은 잘못된 입력 데이타등으로 인하여 스스로 철회될 수 있으며 교착상태(deadlock)나 시스템 고장등으로 인하여 시스템에 의하여 강제로 철회될 수도 있다.

트랜잭션의 개념을 지원하기 위해서는 트랜잭션은 아래와 같은 특성들을 가져야

만 한다 [Gray 81a, Haer 83].

1. 원자성(atomicity): 트랜잭션의 효과가 완전히 반영되거나 전혀 반영되지 않아야 한다.

2. 일관성(consistency): 트랜잭션은 데이타베이스의 일관성을 유지하여야 한다.

3. 지속성(durability): 일단 트랜잭션이 완료되면 그 트랜잭션이 수정한 내용은 이후의 고장에도 불구하고 손실되어서는 않된다.

4. 고립성(isolation): 완료되지 않은 트랜잭션의 실행 결과가 동시에 실행되고 있는 다른 트랜잭션에게 파급되어서는 않된다.

원자성은 트랜잭션의 실행이 고장으로 인하여 방해를 받으면 그 트랜잭션이 수정한 내용은 취소(undo)되어야 함을 의미한다. 뿐만 아니라, 완료된 트랜잭션의 결과는 고장이 발생하여도 복구되어야 한다. 시스템의 여러 가지 고장에 대해 이러한 특성들을 보장하기 위해서는 회복 방법(recovery mechanism)이 필요하다. 트랜잭션의 일관성은 동시성 제어 방법(concurrency control mechanism)을 적용함으로써 보장된다. 고립성은 연속적인 철회(cascading aborts) 문제를 방지함으로써 트랜잭션 개념을 효율적으로 구현할 수 있도록 한다는 점에서 바람직한 특성이다.

4.1.2 동시성 제어

두 개 이상의 트랜잭션들이 동시에 실행되는 경우에 데이타베이스의 일관성을 유지하기 위해서는 동시 실행을 제어하여야 한다. 만약 동시에 실행되는 트랜잭션들이 적절하게 제어되지 않는다면 갱신 손실(lost update)과 비일관성 검색(inconsistent retrieval)등의 이상 현상(anomaly)이 발생한다 [Bern 87]. 동시성 제어는 데이타베이

스를 공유하면서 동시에 실행되는 트랜잭션을 제어하는 것이다. 동시성 제어 알고리즘의 정당성 기준으로서는 순차성(serializability)이 널리 쓰인다 [Bern 79, Papa 79, Papa 86]. 순차성이란 동시에 실행된 결과가 어떤 순차적인 실행 결과와 동일할 때 정확하게 실행되었다고 단정한다.

동시성 제어 방법은 크게 로킹(locking)과 타임스탬프(timestamp), 그리고 낙관적(optimistic) 방법등의 세가지로 분류된다. 로킹 방법 [Crok 86, Eswa 76, Silb 80]에서는 관련된 로크를 얻은 후에 해당 데이타를 액세스 할 수가 있다. 그렇지 않으면 해당 로크를 기다리게 된다. 타임스탬프 방법 [Bern 81]은 각 트랜잭션에 유일하게 할당된 타임스탬프 순서에 따라서 트랜잭션들이 실행된다. 낙관적 방법 [Kung 81, Schl 81]에서는 각 트랜잭션이 끝까지 실행된 후에 일관성 위배 여부를 검토하게 된다.

4.1.3 교착상태

같은 데이타들을 공유하는 두 개 이상의 트랜잭션들이 동시에 실행되는 경우에는 교착상태가 발생할 수가 있다. 교착상태는 트랜잭션들이 연속적으로 서로를 기다리면서 자신들의 실행이 중단되어 있는 상태를 말한다. 동시성 제어 방법으로 로킹 방법을 사용하게 되면 일반적으로 교착상태가 유발된다. 그러나, 트리 로킹 [Kort 82, Silb 80]같은 방법은 교착상태가 발생하지 않는다.

일반적으로, 교착상태를 처리하는 방법은 크게 교착상태 방지(prevention)와 회피(avoidance), 그리고 검출(detection)등의 세가지 방법이 있다. 교착상태 방지는 트랜잭션이 필요로 하는 자원들을 미리 예약하여야 하는 방법이다. 따라서, 트랜잭션이 필요로 하는 모든 자원을 미리 확보하지 않으면 그 트랜잭션은 실행이 허용되지 않

는다. 교착상태 회피는 일단 트랜잭션이 실행을 시작하고 교착상태 발생 가능성이 존재하면 적절한 조치를 취하는 방법이다. 즉, 이미 다른 트랜잭션이 소유하고 있는 자원을 트랜잭션이 요청하게 되면 적절한 조치를 취한다. 교착상태 검출 방법은 교착상태가 발생하도록 허용을 하고 교착상태가 발생되면 검출하여 해결하는 방법이다. 그러므로, 교착상태 검출 방법은 검출(detection)과 해결(resolution)의 두 단계로 이루어 진다. 검출은 보통 대기 그래프에서 사이클을 찾으므로서 이루어 진다. 해결은 교착상태에 관련된 트랜잭션들중에서 희생자(victim) 트랜잭션을 선택하여 철회함으로써 이루어진다.

방지 방법은 교착상태로 인한 트랜잭션의 철회가 없다는 장점이 있다. 그렇지만, 이 방법은 동시성 정도를 격감시키고 트랜잭션들을 미리 분석해야 한다는 단점을 갖는다. 회피 방법은 방지 방법보다는 융통성이 있고 원칙적으로 단순하다는 장점이 있다. 그러나, 이 방법은 필요 이상으로 트랜잭션들이 철회된다는 단점을 가진다. 한편, 검출 방법은 회피 방법보다는 트랜잭션들의 철회가 적다. 그 대신에, 교착상태 검출에 따른 부담과 각각의 트랜잭션들이 동시에 실행되고 있는 다른 트랜잭션들이 소유하고 있는 자원들을 기다려야 하는 단점이 있다.

제 2 절 술어 로킹

술어 로킹(predicate locking)은 데이타베이스를 제기된 질의의 논리 수식 단위로 로킹하여 유령(phantom) 현상을 방지하고 상충성 심사를 위한 부가 노력을 줄이기 위한 방법이다 [Eswa 76]. 술어 로킹에서는 논리 수식을 만족하는 릴레이션의 튜플들만을 로크하기 때문에 만족되지 않는 다른 튜플들을 다른 트랜잭션이 액세스할 수 있다. 따라서, 술어 로킹을 사용함으로써 얻을 수 있는 또 다른 효과는 동시성 정도를 높일 수 있다는 점이다.

데이타베이스 시스템에서 동시성 제어 방법으로 술어 로킹을 사용하는 경우에 해결하여야 하는 가장 큰 문제점은 논리 수식들 간의 상충성 여부를 판단하는 일이다. 임의의 논리 수식에서 이 과정은 일반적으로 쉽게 결정될 수 없으므로 논리 수식을 논리 연산자에 의해 연결된 변수나 상수에 대한 비교 연산자를 $<$, $<=$, $=$, $=>$, $>$, 그리고 $\neq$으로 제한하고 있다. 그러나, 이 경우에도 논리 수식의 상충성 여부를 검토하는 것은 NP-complete 문제가 된다 [Hunt 79]. 따라서, 본 연구에서는 두 논리 수식이 잠재적 상충(potential conflict)의 가능성이 있으면 상충되는 것으로 간주하여 상호 논리 수식의 상충성 검토 문제를 아래와 같이 단순화하여 구현하였다.

1) 논리 수식에 논리 연산자 OR가 존재하면 잠재적 상충으로 간주하였다.
2) 두 논리 수식에 동일한 속성이 나타나지 않으면 잠재적 상충으로 간주하였다.

예를 들어, 두 수식 P_1: $a>c_1$과 P_2: $a<c_2$ OR $b>c_3$를 고려하자. 여기서 a, b는 한 릴레이션의 속성을 나타내며 c_1, c_2, c_3는 임의의 상수를 나타낸다. 이 경우에 P_1과 P_2의 상충성 검토를 위해서는 P_1을 만족하는 튜플들에 대한 속성 b의 값이 P_2에 나타나는 속성 b에 대한 수식을 만족하는지를 검토하여야 한다. 이를 위해서는 실제로 수식 P_1를 만족하는 튜플들을 판독하는 것이 필요하다. 또한, 이것은 본 연구의 프로세스 구조 측면에서 프로세스간의 통신을 의미한다. 프로세스간의 통신으로 야기되는 부가 노력을 줄이기 위하여 이 경우를 잠재적 상충으로 간주하여 구현하였다.

제 3 절 상충성 검사 알고리즘

본 연구에서 두 논리 수식의 상충성을 판정하기 위한 기본 착상은 한 속성에 대한 논리 수식에 대하여 그 속성이 가질 수 있는 최대값(상한)과 최소값(하한)을 구할 수 있다는 데에 있다. 예를 들어, 아래와 같은 수식들을 생각하여 보자.

P_1: $a>3$; (상한,하한)=$(\Omega,3+\epsilon)$

P_2: $a<=10$; (상한,하한)=$(10,-\Omega)$

P_3: $a\neq5$; (상한,하한)=$(\Omega,-\Omega)$

P_4: $2<a<3$; (상한,하한)=$(3-\epsilon,2+\epsilon)$

위의 예에서 Ω는 무한히 큰 가상적인 수를 나타내고 ϵ는 무한히 작은 수를 나타내고 있다.

이제, 한 속성만을 포함하는 두 수식에 대한 비교를 위한 방법을 기술하기로 하자. 두 수식을 P_1과 P_2에 포함된 속성을 a라고 하자. 수식 P_1과 P_2 각각에 대하여 a에 대한 상한과 하한을 위와같이 구할 수 있다. 다음에, P_1과 P_2에 있는 a의 각 상한과 하한을 함께 고려한 병합상한과 병합하한을 구한다. 병합상한과 병합하한은 다음과 같이 정의된다.

* 병합상한: $\min(u_1,u_2)$
* 병합하한: $\max(l_1,l_2)$

여기서 u_i와 l_i는 $(i=1,2)$ 수식 P_i에 있는 속성 a에 대한 상한과 하한을 각기 나타낸다. 따라서, $u_i>=l_i$가 된다. 병합상한과 병합하한은 두 수식을 함께 고려하였을 때에 속성 a갖을 수 있는 최대값과 최소값을 각각 의미한다. 병합하한이 병합상한보다 큰 경우에는 두 수식 P_1과 P_2가 서로 상충되지 않음을 의미한다. 왜냐하면, 병합하한이 된 하한(예로, l_1)에 대응하는 상한(즉, u_1)은 병합하한보다 크거나 같고(즉, $l_1<=u_1$) 병합상한이 된 상한(예로, u_2)에 대응하는 하한(즉, l_2)은 병합상한보다 크지 않고 (즉, $l_2<=u_2$) 또한 병합하한이 병합상한보다 크므로$(l_1>u_2)$ 두 수식 P_1과 P_2의 a에 대한 상한과 하한은 서로 교차되지 않음을 의미하기 때문이다. 즉, 이 경우에 $u_1=>l_1>u_2=>l_2$가 성립된다. 예를 들어, 앞의 수식 P_a, P_b, P_c, P_d를 고려하여 보자. P_a

와 P_b를 고려할 때 속성 a의 병합상한과 병합하한은 각각 10과 $3+\epsilon$가 된다. 병합상한이 병합하한보다 크므로($10>3+\epsilon$) P_a와 P_b는 상충되는 수식이다. 한편, P_a와 P_d를 고려한 속성 a의 병합하한과 병합상한은 각각 $3+\epsilon$와 $3-\epsilon$가 된다. 병합하한이 병합상한보다 크므로($3-\epsilon<3+\epsilon$) P_a와 P_a와 P_b는 서로 충돌이 없는 수식임을 알 수가 있다. 이 알고리즘은 두 개 이상의 논리 수식에도 일반적으로 적용될 수 있음을 알 수 있다

여러개의 속성들을 포함하는 두 수식 P_1과 P_2의 상충성 여부는 P_1과 P_2가 포함하는 각 속성에 대한 상충성 여부를 검토하면 된다. 즉, P_1과 P_2에 나타나는 모든 속성에 대하여 각 속성의 병합하한이 병합상한보다 크면 P_1과 P_2는 서로 상충되지 않는다. 이상으로부터, 본 연구에서 사용한 두 개의 논리 수식을 비교하는 알고리즘은 아래와 같이 요약된다.

입 력: 논리 수식 P_1과 P_2

출 력: 상충성 비교 결과

단계 1): 논리 수식에 OR 연산자가 있는지 검사한다. 존재하면 단계 7)로 가고 존재하지 않으면 단계 2)로 간다.

단계 2): 한 논리 수식에 나타나는 속성들의 집합이 다른 논리 수식에 나타나는 속성들의 집합에 포함되는지 검사한다. 포함되지 않느면 단계 7)로 가고 포함되면 단계 3)으로 간다.

단계 3): 각 속성에 대하여 상한과 하한을 구한다. 단계 4)로 간다.

단계 4): 각 속성에 대하여 병합상한과 병합하한을 구한다. 단계 5)로 간다.

단계 5): 모든 속성의 병합하한이 병합상한보다크면 단계 6)으로 가고 그렇지 않으면 단계 7)로 간다.

단계 6): 상충되지 않음을 출력한다. 단계 8)로 간다.

단계 7): 상충됨을 출력한다. 단계 8)로 간다.

단계 8): 멈춘다.

제 4 절 동시성 제어 프로그램의 구현

본 절에서는 트랜잭션 스케쥴러 프로그램의 구현에 대해 기술한다. 트랜잭션 스케쥴러는 트랜잭션의 시작, 완료, 그리고 철회등과 같은 연산을 구현하기 위해 필요한 트랜잭션의 성질을 제공하고, 여러 트랜잭션이 동시에 실행되는 환경에서 데이타베이스의 일관성을 보장하기 위한 동시성 제어 기법을 제공한다. 본 연구에서 구현된 동시성 제어 프로그램은 본 연구에서 정의된 연산들 중에서 데이타베이스와 화일에 관한 연산, 그리고 데이타 레코드에 관한 연산을 관리한다. 이러한 연산들의 특징은 여러 트랜잭션들이 이 연산들을 동시에 실행할 경우 전체 데이타베이스의 일관성을 유지할 수 없는 경우가 생긴다는 것이다. 트랜잭션 스케쥴러가 선별적인 연산만 관리하는 방식은 동시성 제어가 필요한 연산들을 제외한 다른 연산들은 트랜잭션 스케쥴러를 거치지 않고 질의 처리 프로세스에서 데이타 회복 관리 프로세스로 전달되므로 전체 데이타베이스 시스템에서 전송되는 메세지의 양을 줄일 수 있다는 장점을 갖는다.

4.4.1 데이타베이스와 화일에 관한 연산의 관리

동시성 제어 프로그램이 관리하는 데이타베이스와 화일에 관한 연산은 다음과 같다.

데이타베이스에 관한 연산

kass_mount(DeviceName)

kass_dismount(DeviceName)

화일에 관한 연산

kass_openfile(VolID, FileName)

kass_openindex(VolID, FileName, IndexNo)

kass_openhash(VolID, FileName, HashNo)

kass_destroyfile(VolID,FileName)

kass_dropindex(VolID, FileName, IndexNo)

kass_destroyhash(VolId, FileName, HashNo)

4.4.1.1 데이타베이스에 관한 연산의 관리

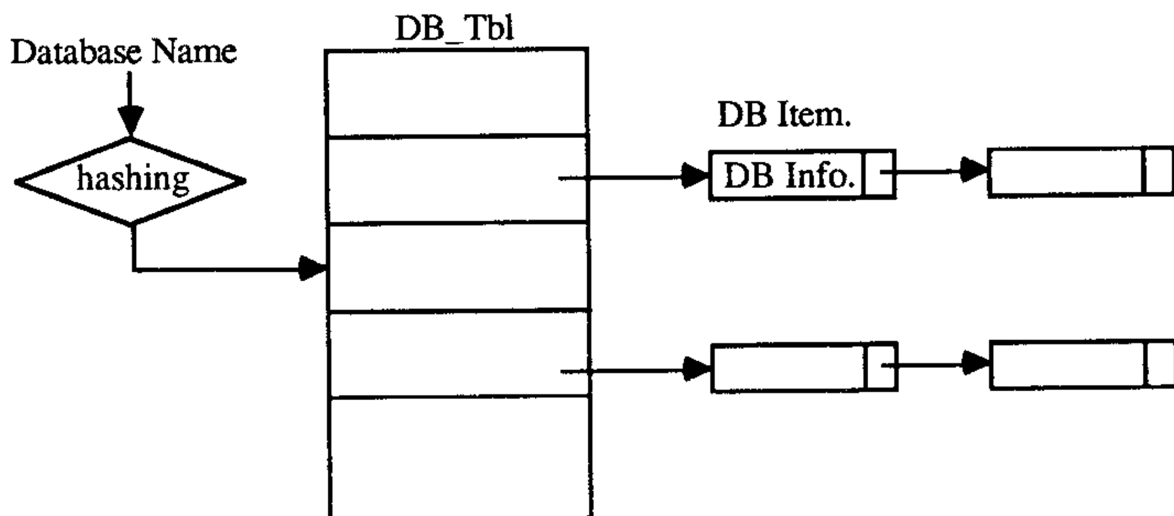


그림 4.1 데이타베이스 테이블

데이타베이스에 관한 연산에 대해 동시성 제어 프로그램이 데이타베이스의 일관성을 유지하기 위해 제공하는 기능은 다음과 같다. 여러 사용자가 데이타베이스를 열어 사용할 경우, 각 사용자가 데이타베이스를 닫더라도 모든 사용자가 데이타베이스를 닫을 때까지 실제적인 데이타이스는 닫쳐서는 않된다. 즉, 데이타베이스를 연 각각의 사용자는 다른 사용자의 데이타베이스에 관한 연산에 관계없이 각 사용자가 연 데이타베이스를 닫을 때까지 사용할 수 있어야 한다. 이를 위해서 트랜잭션 스케쥴러는 데이타베이스의 열기와 닫기를 관리하는 데이타베이스 테이블을 유지하고 그 내용은 그림 4.1과 같다.

그림 4.1에서 나타난 데이타베이스 테이블의 각 항목은 열린 데이타베이스에 해당되고 각 항목의 내용은 그림 4.2와 같다.

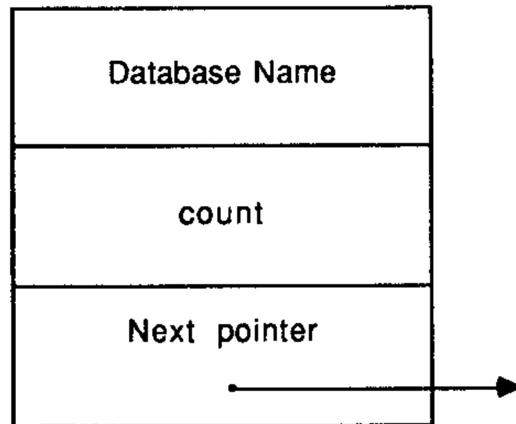


그림 4.2 데이타베이스 테이블의 각 항목의 내용

데이타베이스 테이블의 항목의 내용중에서 count 부분은 데이타베이스를 연 사용자의 수를 나타내고 각 사용자가 닫을 때마다 1씩 감소한다. count의 내용이 0일 때 데이타베이스를 실제로 닫는다. 앞에서 언급한 데이타베이스 테이블의 내용을 관리하는 동시성 제어 프로그램의 부분은 다음과 같다.

check_mount(DeviceName)

check_dismount(DeviceName)

check_mount는 사용자가 데이타베이스를 열 때 실행된다. 데이타베이스가 이미 다른 사용자에 의해 열려 있을 경우 해당되는 데이타베이스 테이블의 항목에 포함된 count의 내용을 1 증가한다. 데이타베이스가 아직 열려 있지 않았을 경우 새로운 데이타베이스 항목을 데이타베이스 테이블에 삽입하고 count를 1로 둔다. check_dismount는 사용자가 데이타베이스를 닫을 때 실행되고, count의 내용을 1 감소한다. count의 내용이 0일 경우 그 데이타베이스를 데이타베이스 테이블에서 삭제한다.

4.4.1.2 화일에 관한 연산의 관리

화일에 관한 연산에 대해 동시성 제어 프로그램이 데이타베이스의 일관성을 유지하기 위해서 제공하는 기능은 다음과 같다. 여러 사용자가 화일을 열어 사용할 경우 다른 사용자가 그 화일을 제거할할 수 없다. 즉, 사용자가 화일을 제거하기 위해서는 그 화일을 열어 사용하고 있는 사용자가 없어야 한다. 마찬가지로, 사용자가 화일을 열어 사용하기 위해서는 그 화일을 제거한 사용자가 없어야 한다. 그러므로 사용자가 한 화일을 제거하기 위해서 현재 현재 화일을 열고 있는 사용자 트랜잭션의 실행이 모두 끝날 때까지 기다려야 한다. 이를 위해서 트랜잭션 스케쥴러는 화일의 열기과 제거를 관리하기 위해 화일 테이블을 유지하고 그 내용은 그림 4.3과 같다.

그림 4.3에서 나타난 화일 테이블의 각 항목은 화일의 열기 혹은 제거에 해당된다. 각 항목은 두 가지 방법에 의해 액세스 될 수 있는데, 화일 이름에 의한 액세스와 그 화일에 대해 연산을 실행한 트랜잭션의 이름에 의한 액세스이다. 화일 이름에 의한 액세스는 그 화일을 처음으로 열기 혹은 제거할 때 이루어지고, 새로 생성된 항목은 그 화일을 포함하고 있는 디스크 볼륨(volume)과 그 화일의 이름을 인자로 하

여 산출되는 해쉬 함수값에 대응하는 위치에 삽입된다. 트랜잭션의 이름에 의한 액세스는 트랜잭션이 완료되었을 때 그 트랜잭션이 액세스한 화일에 대한 항목을 화일 테이블에서 삭제하기 위해서 사용된다. 각 항목의 삭제를 용이하게하기 위해서 각 항목은 쌍방형 연결 리스트로 구현되었다.

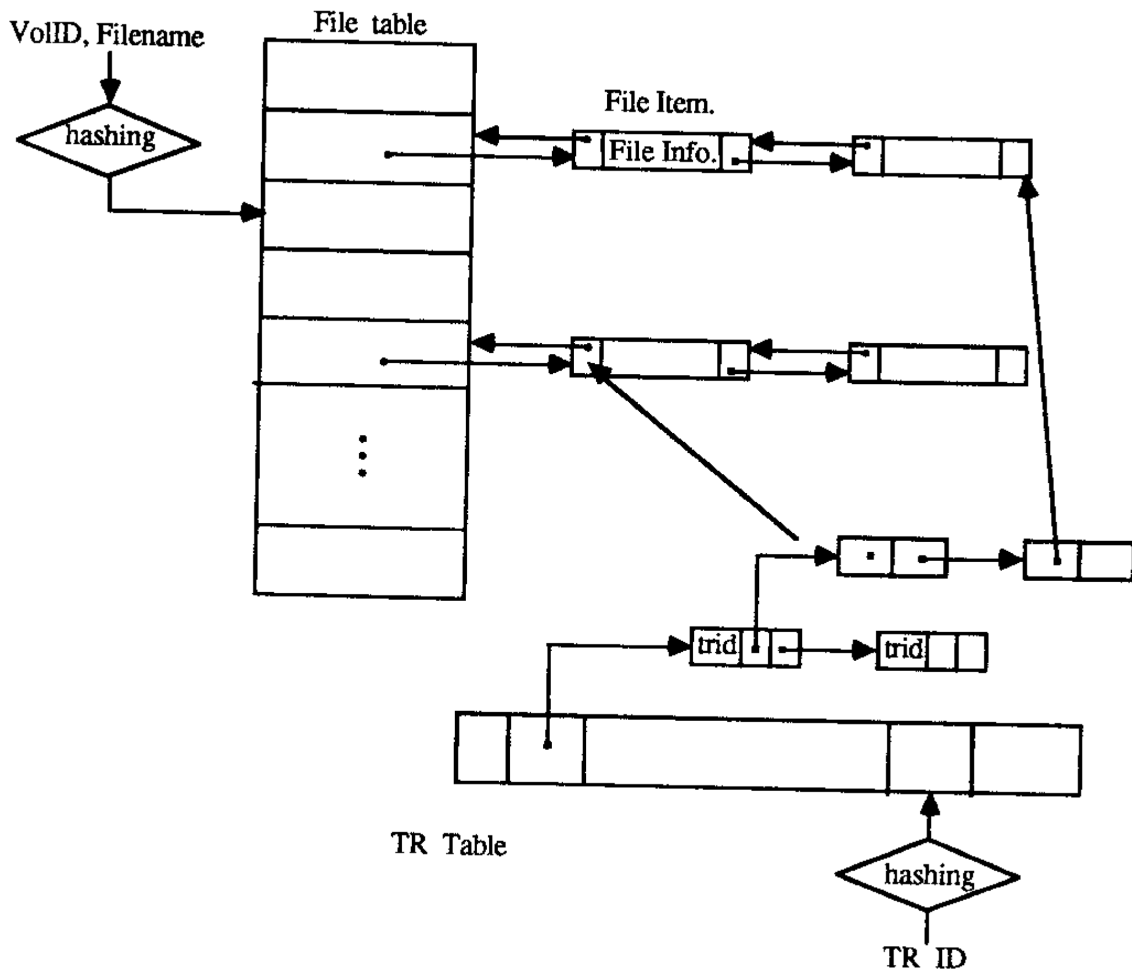


그림 4.3 화일 테이블

본 연구에서 구현된 화일 연산에 대한 관리중에서 주의할 점은 화일에 대한 닫기 연산은 트랜잭션 스케쥴러가 처리하지 않는다는 것이다. 그 이유는 다음과 같은 두 가지 이유로 설명된다. 첫째, 각 트랜잭션이 실행하고자 하는 닫기 연산이 트랜잭

선 스케쥴러를 거치지 않고 바로 데이타 회복 관리자로 전달함으로써 전체 시스템에서 제기되는 메세지 전송량을 줄일 수 있다. 둘째, 트랜잭션 스케쥴러의 관점에서 볼 때 트랜잭션이 이미 연 화일은 그 트랜잭션의 실행이 완료될 때까지 열린 것으로 간주되므로 각 트랜잭션이 액세스하는 화일의 일관성을 유지할 수 있다는 것이다. 즉, 트랜잭션의 실행 과정에서 한 번 열린 화일은 그 트랜잭션이 완료될 때까지 제거될 수 없다. 트랜잭션 스케쥴러가 화일의 닫기 연산을 처리하지 않기 때문에 각 트랜잭션이 완료될 때 열린 화일을 일괄적으로 닫는 과정이 필요하다.

화일 테이블에서 유지되는 각 항목의 내용은 그림 4.4와 같다.

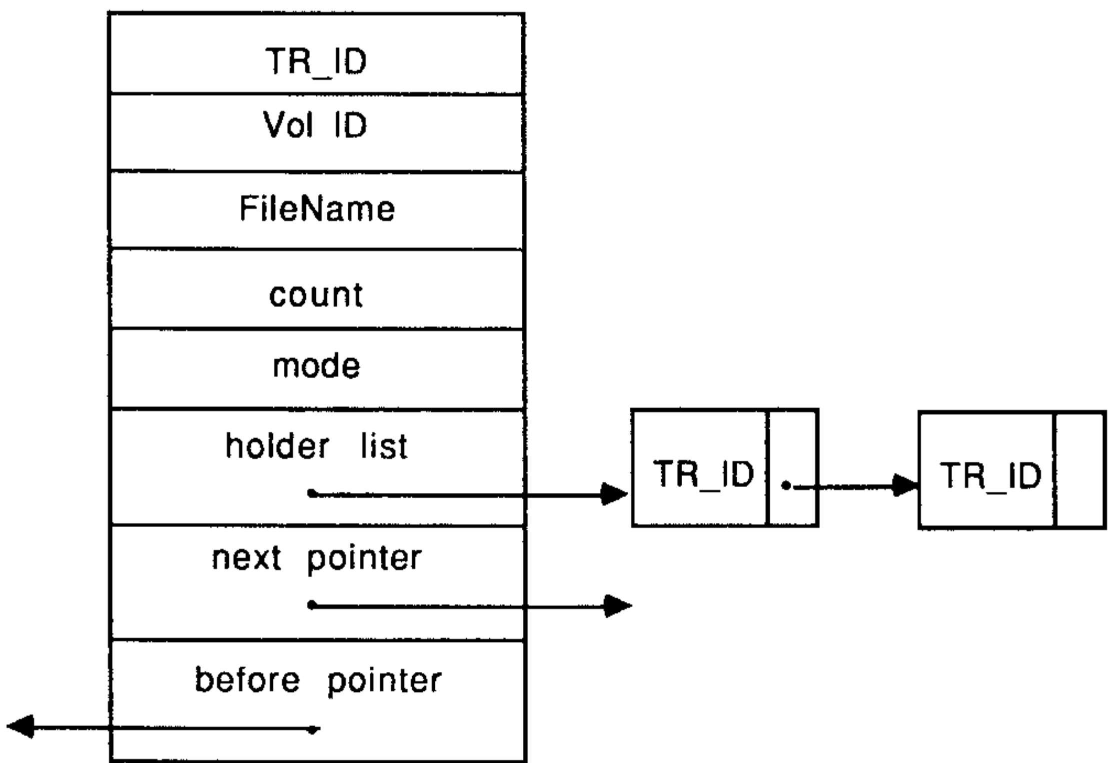


그림 4.4 화일 테이블의 각 항목의 내용

화일 테이블의 항목의 내용중에서 mode는 현재 그 화일 테이블의 상태를 나타내며

열기 혹은 제거의 두 가지 상태를 가진다. count 부분은 현재 mode로 그 화일을 액세스하고 있는 트랜잭션의 수를 나타낸다. 즉, count는 화일이 열기 형태로 액세스될 경우 그 화일을 열고 있는 트랜잭션의 수를 나타내고 제거 형태로 액세스될 경우 1이다. 트랜잭션이 완료할 경우 열기 형태로 액세스한 화일의 count를 1씩 감소한다. count가 0일 때 그 항목은 화일 테이블에서 삭제된다. 0이 아닐 경우, 항목의 Tid는 holder에 유지되고 있는 리스트의 첫번째 원소로 지정된다.

4.4.2 데이타 레코드에 관한 연산의 관리

데이타 레코드에 관한 일반적인 연산을 처리하기 위한 동시성 제어 프로그램은 스케쥴러 프로그램, 로크 관리 프로그램, 그리고 교착 상태 검출 프로그램으로 구성된다. 스케쥴러 프로그램에서는 로크표를 관리하는 로크 관리 프로그램 에게 로크를 요청하고, 로크가 허락되면 데이타 회복 관리자에게 그 연산의 실행을 요청한다. 로크가 거절될 경우 교착 상태 검출 프로그램에게 교착 상태 발생 여부를 검사하게 한 후, 교착 상태가 발생하면 희생자 트랜잭션을 철회한다. 트랜잭션 스케쥴러가 질의 처리 프로그램으로부터 받아서 처리하는 연산은 모두 스캔 연산이다. 그 이유는 데이타 레코드를 액세스하기 위해 액세스하고자 하는 데이타 레코드를 지정하는 스캔 연산이 선행되어야 하고 실제 데이타 레코드에 대한 판독/기록 연산은 스캔 연산에 의해 지정된 데이타 레코드에만 실행되기 때문이다. 그러므로 4.2절과 4.3절에서 설명한 술어 로킹 알고리즘을 각 스캔 연산에 적용함으로써 전체 시스템의 동시성을 제어할 수 있다는 것이 본 연구에서 사용된 기본 개념이다. 데이타 레코드에 관한 연산중에서 트랜잭션 스케쥴러가 처리하는 연산의 종류는 다음과 같다.

kass_openfilescan(OpenFileNum, BoolExp, scan_mode)

kass_openindexscan(OpenFileNum, IndexFileNum, IndexKey, LB, UB,

BoolExp, scan_mode)

kass_openhashscan(OpenFileNum, IndexFileNum, IndexKey, SearchKey,

BoolExp, scan_mode)

각각의 스캔 연산은 데이타 화일에 대한 스캔 연산과 인덱스 화일에 대한 스캔 연산으로 분류된다. 데이타 화일에 대한 스캔 연산은 해당되는 데이타 화일에만 로크를 유지하면 되지만, 인덱스 화일에 대한 스캔 연산은 인덱스 화일에 대한 로크와 인덱스 화일에 대응되는 데이타 화일에도 로크를 유지해야 한다. 그 이유는 본 연구에서 제공되는 인덱스 화일에 대한 스캔 연산의 의미는 인덱스 화일을 스캔하여 얻은 정보를 이용하여 데이타 화일의 레코드를 액세스한다는 것을 나타내기 때문이다. 데이타 화일에 대한 스캔 연산을 처리하는 기능은 다음 함수에 의해 정의된다.

```
data_execute(Tid, Fnum, Exp, Uexp, Mode, in_msg)
int Tid, Fnum, Mode;
BOOLEXP *Exp, *Uexp;
structure message *in_msg;
```

Exp는 스캔에서 사용되는 술어이고, Uexp는 갱신 연산일 경우 갱신할 애트리뷰트의 값들을 나타내는 술어이다. Mode는 스캔 연산의 열기 형태로서 read 혹은 write의 값을 갖는다. data_execute를 실행한 결과가 ACCEPT일 경우 in_msg의 내용이 데이타 회복 관리자에게 전달된다. 그렇지 않을 경우 data_execute를 실행한 트랜잭션은 대기 상태가 된다. 마찬가지로 인덱스 화일에 관한 스캔 연산을 처리하는 기능은 다음 함수에 의해 정의된다.

```
index_execute(Tid, I_Fnum, D_Fnum, Exp, Uexp, Mode, in_msg)
int Tid, I_Fnum, D_Fnum, Mode;
```

```
BOOLEXP *Exp, *Uexp;
structure message *in_msg;
```

동시성 제어에 관한 자세한 내용은 다음절에서 설명한다.

4.4.2.1 스케쥴러 프로그램

각각의 스캔 연산에 대해 스케쥴러 프로그램은 로크 관리 프로그램에게 로크를 요청하며, 로크 요청에 대한 로크 프로그램의 응답에는 ACCEPT와 REJECT가 있다. ACCEPT시 로크 관리 프로그램은 로크 정보를 유지하는 원소를 로크표에 저장한다. 이때 그 원소가 적용되는 (볼륨 이름, 화일 이름)의 해쉬 함수값을 갖는 Open_File_Table의 위치에 연결하고, 마찬가지로 그 원소를 실행한 트랜잭션의 (트랜잭션 식별자)의 해쉬값을 갖는 TR_Table의 위치에 연결한다. 교착 상태 검출 프로그램은 로크 관리 프로그램에 의해 REJECT가 반환될 경우 스케쥴러에 의해 호출되며 교착 상태의 발생 여부를 검사한다. 교착 상태가 발생하면 교착 상태 검출 프로그램은 희생자를 선정하여 스케쥴러 프로그램에게 반환하고, 교착 상태가 발생하지 않으면 대기 리스트에 그 연산에 관한 정보를 저장한다.

스케쥴러 프로그램은 데이타 화일에 대한 스캔 연산과 인덱스 화일에 대한 스캔 연산에 대해 서로 다른 동시성 제어 기법을 제공한다. 그 이유는 앞에서 언급했듯이 인덱스 화일에 대한 스캔 연산은 데이타 화일에 대한 액세스도 포함하고 있기 때문이다. 그러므로 인덱스 화일에 대한 스캔 연산을 처리하기 위해서는 데이타 화일에 대한 새로운 스캔 연산을 정의하여야 하며, 각각의 스캔 연산에 대한 로킹이 허용될 경우에만 인덱스 화일에 대한 인덱스 화일에 대한 스캔 연산을 실행할 수 있다. 데이타 화일과 인덱스 화일에 대한 스캔 연산을 실행하기 위해 본 연구에서 구현된 함수는 다음과 같다.

```
int data_schedule(Tid, OpenFileNum, Bexp, Uexp, Mode, victim)
int Tid, *victim;
int OpenFileNum, Mode;
BOOLEXP *Bexp, *Uexp;

int index_schedule(Tid, I_Num, D_Num, Bexp, Uexp, Mode, victim)
int Tid, *victim;
int I_Num, D_Num, Mode;
BOOLEXP *Bexp, *Uexp;
```

본 연구에서는 인덱스 화일에 대한 스캔 연산을 처리하기 위해 인덱스 화일의 스캔 연산에 대한 로킹을 신청하고 허용될 경우 데이타 화일의 스캔 연산에 대한 로킹을 신청한다. 데이타 화일의 스캔 연산에 대한 로킹이 허용되지 않을 경우, 인덱스 화일에 대한 스캔 연산은 실행될 수 없고 인덱스 화일에 대한 스캔 연산을 실행하려고 하는 트랜잭션은 대기 상태로 변환된다. 이때 이 트랜잭션이 먼저 보유하고 있던 인덱스 화일의 스캔 연산에 대한 로킹은 철회된다. 그 이유는 인덱스 화일의 스캔 연산에 대한 로킹을 철회하지 않을 경우 동일한 환경에 있는 다른 트랜잭션과의 교착상태 발생의 확률이 증가하기 때문이다. 본 연구에서 구현된 술어 로킹과 같이 로킹되는 데이타 단위가 큰 환경에서는 교착상태 발생의 확률이 데이타 단위가 작은 환경보다 크므로, 가능하면 교착상태 발생 확률을 줄이려는 노력이 필요하다. 인덱스 화일의 스캔 연산에 대한 로킹을 철회함으로써 교착상태가 발생할 가능성은 줄었지만, 인덱스 화일에 대한 스캔 연산을 재실행할 때마다 두 번의 로킹 검사가 필요하다는 단점이 있다.

4.4.2.2 로크 관리 프로그램

로크 관리 프로그램은 4.2절과 4.3절에서 설명한 술어 로킹 알고리즘을 구현한 것으로서 로크 테이블을 관장하며 각 술어들간의 상충성을 판단한다. 로크 테이블의 전체 구조는 그림 4.5와 같다.

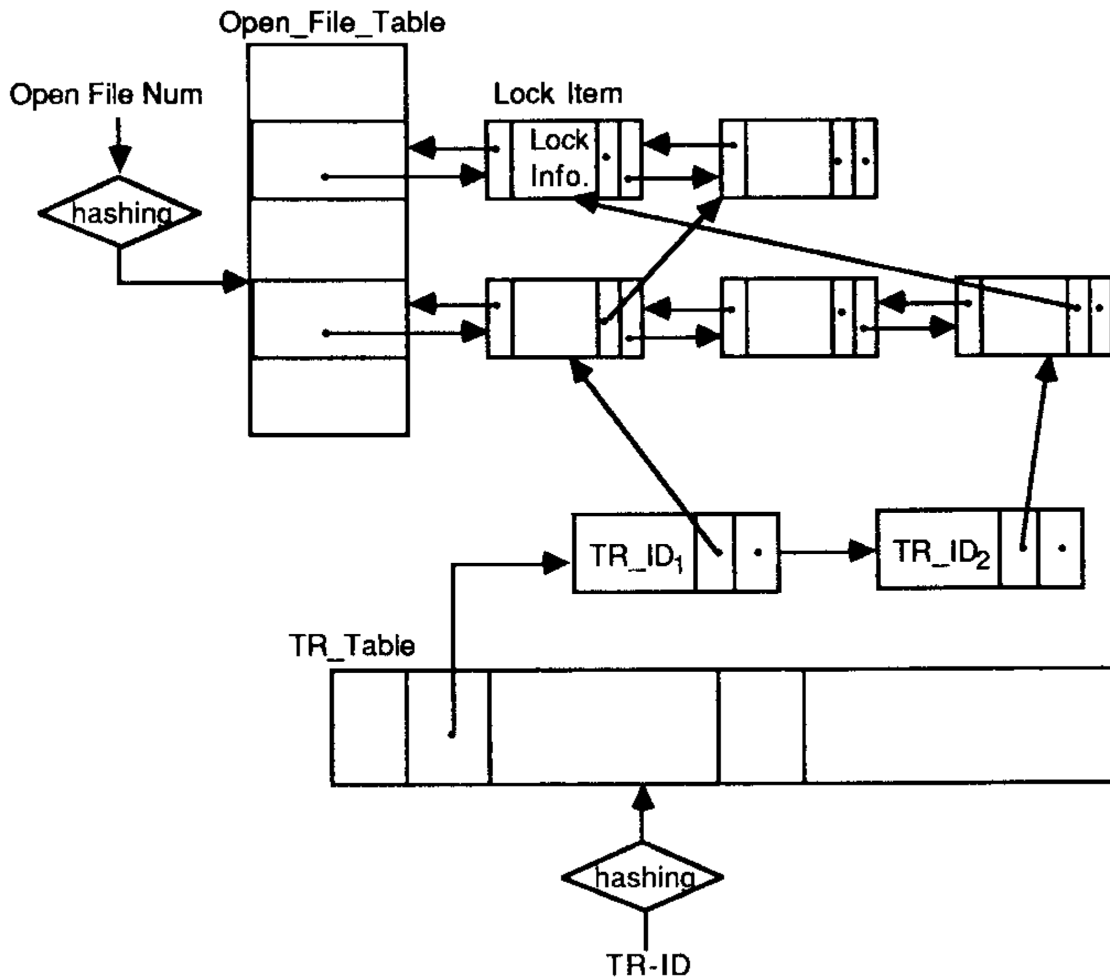


그림 4.5 로크 테이블의 구조

로크 테이블의 각 항목은 스캔 연산이 실행될 때마다 생성된다. 그리고 로크 테이블의 각 항목은 두 가지 방법에 의해 액세스될 수 있는데, 화일 이름에 의한 액세스와 그 화일에 대해 연산을 실행한 트랜잭션의 이름에 의한 액세스이다. 화일 이름에 의한 액세스는 그 화일에 포함된 데이타 레코드를 액세스하기 위해 스캔 연산을

실행할 때 이루어지며, 새로 생성된 로크 항목은 스캔 연산이 수행되는 열린 화일의 이름(OpenFileNum)을 인자로 하여 산출되는 해쉬 함수값에 대응하는 위치에 삽입된다. 트랜잭션의 이름에 의한 액세스는 트랜잭션이 완료되었을 때 그 트랜잭션이 실행한 스캔 연산에 관한 정보를 삭제하기 위해 사용된다. 각 항목의 삭제를 용이하게하기 위해서 각 항목은 쌍방형 연결 리스트로 구현되었다. 로크 테이블에서 유지되는 각 항목의 내용은 그림 4.6과 같다.

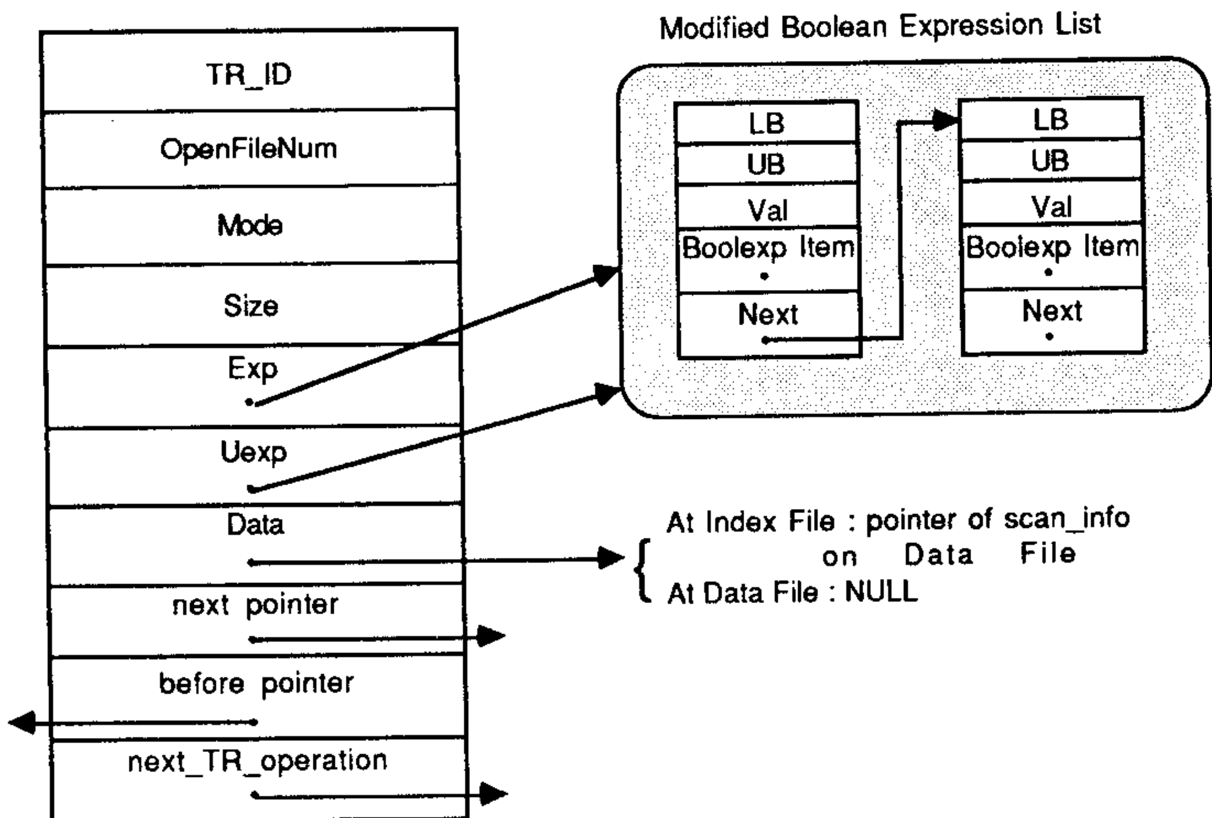


그림 4.6 로크 테이블의 각 항목의 내용

로크 테이블의 항목의 내용중에서 Mode는 스캔 연산을 실행한 트랜잭션이 보유하고 있는 로킹의 내용을 나타내며 read 혹은 write의 값을 갖는다. 본 연구에서 구현된 술어 로킹은 3장에서 설명한 Boolean Expression에 기초를 두고 있다. 그러므

로 로크 테이블의 각 항목에서는 스캔 연산에 관련된 Boolean Expression에 관한 정보를 유지하고 있는데, Size는 Boolean Expression를 구성하고 있는 각 술어의 수를 나타내고 있으며 Exp는 로크 테이블의 항목이 유지하고 있는 Boolean Expression에 대한 포인터이다. Exp에 의해 유지되는 정보는 각각의 Boolean Expression에 대해 4.3절에서 설명한 병합 상한(UB)와 병합 하한(LB)의 값, 그리고 Boolean Expression에서 사용된 값의 형을 double로 형 변환(type casting)한 값(Val) 등이다.

데이타 레코드를 갱신할 목적으로 스캔 연산이 사용될 경우에 Uexp는 갱신하고자 하는 애트리뷰트와 그 애트리뷰트의 값을 나타내는 Boolean Expression의 형태로 나타난다. 갱신 연산을 실행하기 위해서는 Exp를 이용하여 스캔 연산의 허용 여부를 검사하고, Uexp를 사용하여 갱신 연산의 실행이 다른 스캔 연산에 영향을 미치지 않음을 보장하는 절차가 필요하다. Uexp가 정의되지 않을 경우 갱신된 후의 값이 다른 스캔 연산에 영향을 미칠 수 있으므로 전체 데이타베이스의 일관성을 보장할 수 없다는 문제점이 나타난다. 그러므로 갱신 연산을 실행하기 위해서는 Exp와 Uexp에 대한 두 가지 술어의 로킹이 허락되어야 한다. 그렇지 않을 경우 갱신 연산을 실행하고자 하는 트랜잭션은 갱신 연산을 실행할 수 없고 대기 상태로 상태 변환을 한다.

앞에서 언급했듯이, 화일에 대한 스캔 연산에는 데이타 화일에 대한 스캔 연산과 인덱스 화일에 대한 스캔 연산이 있다. 인덱스 화일에 대한 스캔 연산일 경우 Data는 데이타 화일에 대한 로크 테이블의 항목의 정보를 유지하는 포인터로 사용되고, 데이타 화일에 대한 스캔 연산일 경우 Data는 NULL이다.

4.4.2.3 교착상태 검출 프로그램

교착상태를 검출하기 위해서는 트랜잭션들이 로크 허용을 기다리고 있는 상태에 대한 정보를 유지하여야 한다. 이 정보를 나타내고 있는 것을 대기 그래프라 한다.

본 논문에서 구현된 대기 그래프의 구조는 그림 4.7과 같다.

그림 4.7에서 List_of_waiter는 Tid를 기다리고 있는 트랜잭션의 리스트를 나타낸다. 스케쥴러 프로그램의 DeadlockDetect 호출로부터 트랜잭션 T_i가 트랜잭션 T_j를 기다린다는 정보를 받으면 교착상태 검출 프로그램은 교착상태의 검출을 위해 다음과같은 단계를 거친다.

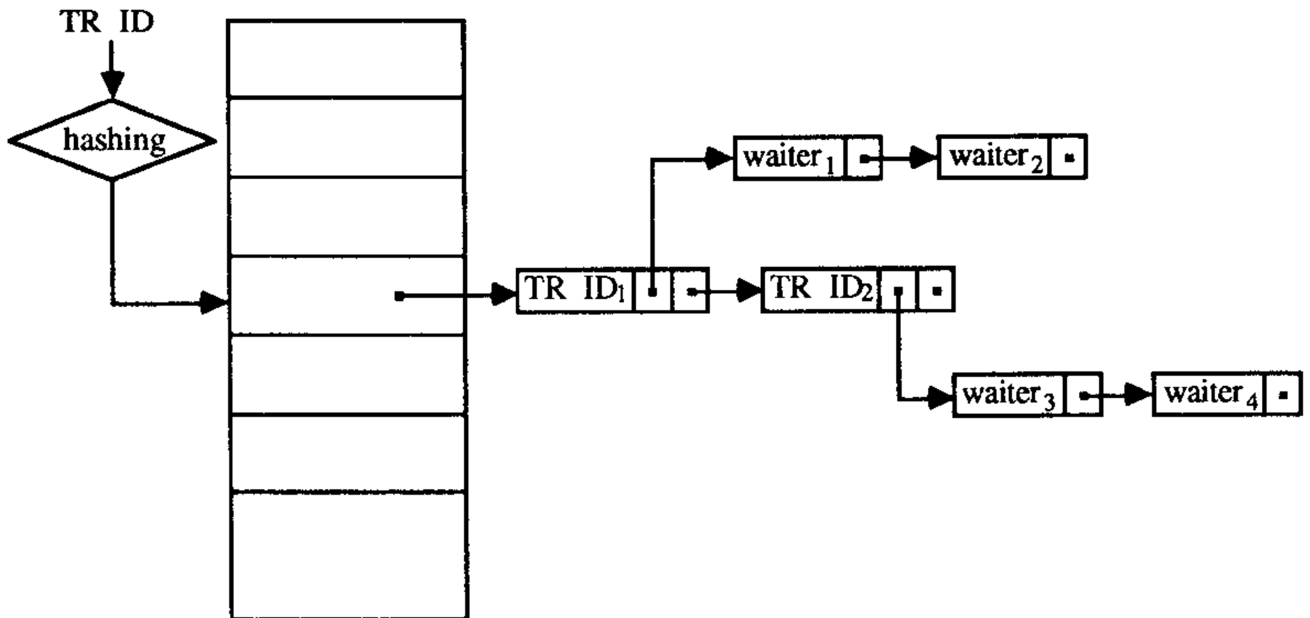


그림 4.7 대기 그래프의 구조

1. 대기 그래프에서 T_i의 대기 정보를 유지하는 노드 n_i를 해쉬 함수를 이용하여 찾는다. 즉, 대기 그래프에서 연결 리스트 노드의 Tid 필드가 T_i와 같은 노드를 찾는다. 노드 n_i가 존재하지 않으면 T_i를 T_j의 노드 n_j의 List_of_waiter에 첨가하고 단계 3으로 가고 노드 n_i가 존재하면 단계 2로 간다.
2. 노드 n_i의 List_of_waiter에 포함된 모든 트랜잭션 T_k에 대해 $(k = j)$이면 교

착상태가 발생한 상태이므로 단계 3으로 간다. (k = j)인 T_k가 존재하지 않을 경우 각 트랜잭션 T_k의 노드 n_k의 List_of_waiter에 T_j가 존재하는 가를 순환적으로 검사하여 교착상태가 발생할 경우 단계 3으로 간다. 모든 트랜잭션 T_k에 대해 교착상태가 발생하지 않을 경우 T_i를 n_j의 List_of_waiter에 첨가하고 단계 3으로 간다.

3. 교착상태가 검출되었으면 교착상태 해결을 위하여 트랜잭션 T_i를 철회하고 교착상태가 아니면 교착상태 검출을 끝낸다.

트랜잭션 철회에 관한 자세한 설명은 다음절에서 계속된다.

4.4.3 트랜잭션 종료 관리 프로그램

트랜잭션 종료 관리 프로그램은 트랜잭션이 완료 혹은 철회될 때 호출된다. 트랜잭션 종료 관리 프로그램의 호출문 형태는 다음과 같다.

restart_and_abort(T_id)

트랜잭션이 종료될 경우, 트랜잭션 T_id가 유지하고 있던 모든 로크 정보를 화일 테이블과 로크 테이블에서 삭제하고 대기 상태에 있던 연산들을 재실행한다. 화일 테이블에서 T_id가 액세스하고 있던 화일에 대한 정보를 제거하기 위해 T_id의 해쉬 함수값에 대응하는 TR_Table의 위치를 파악하고, TR_Table이 가리키는 항목들의 리스트를 따라가면서 화일 테이블의 각 원소들을 삭제한다. 로크 테이블에서 로크 정보의 삭제도 이와 유사하게 이루어진다. 트랜잭션이 실행한 모든 연산에 대한 로크를 삭제한 후, 트랜잭션 종료 관리 프로그램은 그 트랜잭션에 의해 대기 상태에 있던 다른 연산들을 재실행한다. 이때 각 연산의 재실행은 연산이 처음 실행된 것과 동일한 과정을 거친다. 트랜잭션 T_id가 유지하고 있던 모든 테이블에 대한 삭제와 대기

상태에 있던 모든 연산들의 재실행을 위해 다음과 같은 프로시쥬어가 호출된다.

File_OP_Terminate(T_id, &Tlist, &Alist)

Transaction_Terminate(T_id, &Tlist, &Alist)

Tlist는 트랜잭션 T_id가 종료됨에 따라 실행이 보장된 트랜잭션들의 리스트이고, Alist는 T_id가 종료되어 재실행한 결과로서 철회될 트랜잭션들의 리스트이다. 트랜잭션 종료 프로그램은 Tlist에 포함된 각각의 트랜잭션이 실행하고자 하는 연산들을 데이타 회복 관리자에게 전달한다. 그리고 Alist에 포함된 각각의 트랜잭션을 철회한다. 이때 각 트랜잭션의 철회는 트랜잭션 T_id의 종료와 동일한 과정을 거친다. 트랜잭션을 철회할 때 그 트랜잭션의 철회 사실을 데이타 회복 관리자에게 전달하고, 데이타 회복 관리자는 트랜잭션의 철회에 따른 처리를 한 후 트랜잭션의 철회를 질의 처리 프로세스에게 알린다.

제 5 장 데이타 회복 관리

본 장에서는 데이타 회복 관리 프로그램 (Data Recovery Manager:약칭 DBM)의 설계와 구현에 대해 기술한다. 시스템 혹은 트랜잭션은 여러가지 요인으로 인하여 고장이 발생할 수 있다. 여러가지 고장으로부터 시스템의 회복을 위해서는 시스템이 정상적으로 동작할 때에 여러가지 정보를 기록하여 두는 준비 단계와 고장이 발생할 때 이를 회복하는 회복 단계를 거쳐야 한다. 본 연구에서는 트랜잭션의 철회(undo)와 재수행(redo)을 요구하는 회복 알고리즘을 사용하며, 이를 위하여 로그(log)를 이용한다.

제 1 절 고장의 형태

시스템의 회복 관리 프로그램을 설계하기 위해서는 예상되는 고장의 형태를 구분하여야 한다. 데이타베이스 시스템에서 중요하게 고려되는 고장의 종류에는 다음의 세 종류가 있다[Ber 87].

- 트랜잭션 고장
- 시스템 고장
- 보조 기억장치 고장

위의 고장중에서 시스템 고장이나 보조 기억장치 고장에 대한 검출은 운영체제나 시스템 운영자(operator)가 한다고 가정한다. 그러나, 트랜잭션 고장은 트랜잭션 관리 프로그램 혹은 질의 처리 시스템이 할 수 있다.

트랜잭션 고장은 트랜잭션 내의 오류로 인하여 발생되는 경우와 교착상태를 해결할 때 스케줄러에 의하여 어느 트랜잭션이 희생자로 선정되는 경우가 있다. 이 경우에는 트랜잭션이 실행되기 이전의 일관성 있는 상태로 데이타베이스를 회

복시켜야 한다. 즉, 이런 경우에 시스템은 그 트랜잭션을 철회하여야 한다.

시스템 고장은 운영체제의 고장이나 하드웨어의 고장등으로 인하여 시스템이 정상적으로 트랜잭션을 계속 처리할 수 없게 되는 경우로 주기억장치의 내용이 손실되는 경우가 된다. 따라서, 실행이 완료되지 않은 트랜잭션은 트랜잭션의 고장과 같이 철회되어야 하고, 이미 완료된 트랜잭션은 트랜잭션의 지속성(durability)을 보존하도록 재수행되어야 한다. 이를 위해서는 로그를 검색하여 실행이 완료된 트랜잭션과 완료되지 않은 트랜잭션을 구별하여야 한다.

보조 기억장치의 고장은 시스템 고장과는 달리 보조 기억장치 내용이 손실된다. 이 기억장치 고장의 경우에 회복은 안전 기억장치로부터 데이타베이스를 이전의 일관적 상태로 회복시켜야 한다. 실제로 완벽한 안전 기억장치는 존재하지 않으므로 여러개의 보조 기억장치에 데이타베이스를 중복하여 저장함으로써 안전 기억장치의 효과를 얻고 있다. 많은 시스템에서는 디스크와 마그네틱 테이프에 데이타를 중복되게 저장함으로써 안전 기억장치를 구현한다.

제 2 절 고장 회복 방법들

앞 절에서 언급한 여러가지 고장으로 부터 회복하는 방법은 많이 있으나, 특히 본 연구와 관련이 있는 방법들을 중점적으로 다룬다.

5.2.1 트랜잭션 고장으로부터의 회복

트랜잭션의 고장은 트랜잭션이 실행되기 이전의 일관성 있는 상태로 데이타베이스를 회복시켜야 한다. 즉, 이런 경우에 시스템은 그 트랜잭션을 철회하여야 한다.

데이타 회복 관리 프로그램이 데이타베이스의 각 데이타 요소들을 유지하는

방법에 따라 고장 회복 방법도 달라졌다. 데이타베이스를 유지하는 방법에는 각 데이타 요소를 하나의 안정된 기억장치에 정확하게 유지하는 것과 둘 이상의 안정된 기억장치 안에 유지하는 것이 있다. 즉, 전자를 *in-place updating* 방법이라 하여 각 데이타 요소의 내용이 변경될 때마다 매번 이전의 값이 파괴되는 경우(오직 새로이 변경된 내용만 유지)이고, 후자는 쉐도우 사본(*shadow copy*) 방법이라 하여 각 데이타 요소의 내용들에 대하여 하나 이상의 사본을 가지고 있어 해당 데이타 요소의 이전 내용들이 파괴됨이 없이 안정된 기억장치에 유지되도록 관리하는 경우이다. 이때 이전의 데이타의 내용(old version)을 쉐도우 사본이라고 부른다.

전자의 경우와 같은 방법을 구현하기 위하여 안전한 기억장치에 변경된 내용을 기록하는 로깅 방법(loging)이 사용되며, 후자의 경우를 위하여 쉐도우 페이징(shadow paging) 방법이 사용된다([Gra 81],[Ber 87],[Reu 82]).

가. 로깅 방법

트랜잭션에 고장이 발생하는 경우에 트랜잭션의 철회는 그 트랜잭션이 실행 중에 변경한 내용을 변경되기 전의 값으로 대체하여야 한다. 이러한 작업은 안전한 기억장치에 기록되어 있는 회복 정보를 사용한다. 이러한 이유 때문에 통상 DRM은 안전한 기억장치 안에 데이타베이스 그 자체와 부수적인 정보를 저장해 놓는다. 이러한 부수적인 정보 중의 하나가 로그이다.

개념적으로 로그는 트랜잭션들의 실행과정을 기술하고 있다. 이러한 로그의 형태는 물리적인 로그와 논리적인 로그가 있다. 물리적인 로그는 트랜잭션에 의해 쓰여진 데이타 요소의 값에 관한 정보들을 포함하고 있는 로그의 형태이다.

이러한 로그의 구조는 트랜잭션, 데이타 요소, 및 변경된 내용 등의 3가지 요소로 구성된 [T_i, x, v] 형태로 구성하게 된다. 로그의 자료구조인 로그 레코

드에 각 데이타 요소의 마지막으로 완료된 내용을 포함하고 있어, 트랜잭션의 철회는 위와 같이 기록된 로그 정보를 사용한다. 이를 위해 로그의 기록은 쓰기 연산(write)이 일어났던 순서로 작성되어야 한다. 이러한 정보를 기록하기 위한 한가지 쉬운 방법은 로그를 순차 화일(sequential file)로 하고, 로그 안에 있는 요소들을 그들과 상응하는 쓰기 연산의 순서와 일치하도록 함으로써 이루어진다. 따라서 로그 안에서 [T_i, x, u]가 [T_j, x, v]에 대하여 선행되기 위해서는 W_i[x]가 W_j[x]전에 수행되어야 한다.

로그 안에 데이타베이스 안에 쓰여진 내용을 포함하는 것 대신에 명령어들에 대한 높은 수준의 내용을 포함할 수 있다. 이러한 로깅을 논리적인 로깅이라 한다. 예를들어, 논리적인 로깅의 한 로그 레코드는 "레코드 r은 화일 F에 삽입되었고, F의 색인들은 이러한 삽입 연산을 발생하기 위해 변경되었다."의 형태를 갖는다. 논리적인 로깅의 사용으로 화일과 그 자체의 내용들의 실질적인 쓰기 연산에 상응하는 여러 로그 레코드들 대신에 명령어들의 내용을 기록함으로써 소수의 로그 항목들만 필요시된다. 이러한 형태의 로그 감축으로 시스템의 성능을 향상시킬 수 있으나, 트랜잭션의 철회과정에서 로그 정보를 해석하여야 하는 부수적인 복잡성이 존재한다. 따라서 일반적으로 물리적인 로깅 방법을 사용한다.

나. 쉐도우 페이징 방법

각 데이타 요소에 대하여 맨 마지막으로 완료된 내용은 안정된 기억장치 안에 저장된 영역 안에 기록된다. 버전을 관리하기 위하여 안정된 기억장치 내에는 임의의 데이타 요소들의 완료되지 않았던 버전들을 가리키고 있는 디렉토리를 포함하고 있다. 이러한 디렉토리의 내용은 정상적으로 로그 안에 저장될 변경전의 내용과 변경 후의 내용 모두를 포함하고 있다.

트랜잭션 T_i가 한 데이타 항목 x를 변경할때에 x의 새로운 버전(변경된 내용)은 안정된 기억장치 안에 만들어진다. 현재 수행 디렉토리는 T_i에 변경된 새로운 버전을 지칭하도록 변경된다. 결과적으로 새로운 버전은 T_i가 완료될 때까지 로그의 일부가 된다. T_i 완료될때에 완료되었던 데이타베이스 상태들을 가지고 있던 디렉토리는 T_i에 의해 변경된 버전들을 가리키도록 수정되어야 한다. 이것은 T_i의 변경 결과가 T_i의 완료에 의해 새로운 완료된 데이타베이스 상태가 되는 것을 의미한다. 이러한 구조를 위하여 안정된 기억장치 안에 완료된 데이타베이스를 유지하는 변경된 상태 디렉토리(shadow directory), 현재 동작 중인 상태 디렉토리(current directory)와 현재 동작 중인 내용(scratchy copy)를 유지해야 한다.

트랜잭션 T_i가 완료될때에 변경된 내용이 있는 경우에 회복 관리 프로그램은 x을 위한 현재 수행 중인 디렉토리의 내용을 x의 새로운 버전을 가리키도록 변경시킨다. T_i가 변경하지 않은 데이타 요소에 대해서는 현재 수행 중에 디렉토리의 내용은 변경된 상태 유지 디렉토리의 내용과 일치한다. 그러나 이 두 디렉토리의 변경은 한 원자적 연산에 의해 이루어져야 한다. 이러한 원자적 연산은 안정된 기억장치 내에 있는 마스타 레코드에 의해 이루어진다. 쉐도우 페이징 기법이 사용되는 실례를 설명하면 다음과 같다.

두 데이타 요소 x와 y가 T_i에 의해 종료되는 경우 트랜잭션은 새로운 버전을 만들고 x와 y의 이전 버전은 쉐도우 사본이 된다. T_i는 종료된 후에 현재 수행 중인 디렉토리가 안정된 데이타베이스를 반영할 수 있도록 만든다. 이때 마스타 레코드는 그 내용을 새로운 디렉토리를 가리키도록 변경된다. 5.2.2 시스템 고장으로부터의 회복

트랜잭션의 실행과정에서 어떤 데이타 요소를 안정된 기억장치에 *flush*하도록 강요하는 시점에서 데이타 영역의 교체을 위한 필요한 전략을 사용한다. 이러한

*flush*는 안정된 데이타베이스와 로그에 쓰여진 내용에 따라 이루어진다. 따라서 시스템 고장이 발생된 경우에 복구를 위해 동작되는 시스템 재실행(restart) 명령어는 항상 안정된 기억장치 안에서 필요한 정보들을 찾게 되고, 이러한 내용들이 안정된 데이타베이스 또는 로그이다.

일반적으로 시스템 고장과 연관되어 *'undo'*와 *'redo'*라는 용어를 사용한다. 만약 DRM이 완료되지 않았던 트랜잭션이 안정된 데이타베이스 안에 변경된 값을 기록하는 것을 허락한 경우에서 시스템 고장이 일어난 경우에 DRM은 undo를 요구한다. 즉, 이 시점에서 시스템 고장이 일어나면 안정된 데이타베이스는 완료되지 않았던 트랜잭션의 결과를 포함하고 있을 것이다. 이러한 결과는 안정된 데이타베이스가 잘못된 실행에 관련된 것이므로 그 자체의 내용이 완료 상태로 원상 회복되도록 시스템 재실행에 의해 undo되어야 한다. 만약 트랜잭션의 수행에 의해 변경된 모든 값들이 안정된 데이타베이스 안에 기록되기 전에 트랜잭션이 완료하는 것을 허락한 경우, DRM은 redo를 요구한다.

안정된 데이타베이스 안에 변경된 값을 기록하는 것에 연관시켜 트랜잭션의 완료 순서를 규칙적으로 함으로써 DRM은 redo 또는 undo를 관리할 수 있다. 고장으로부터의 회복을 관리하기 위하여 아래의 4가지 형태의 알고리즘으로 구분된다.

(1) undo와 redo 모두 요구하는 경우

(2) undo만 요구하는 경우

(3) redo만 요구하는 경우

(4) undo와 redo 모두 요구하지 않는 경우

고장 회복 알고리즘에서 undo 또는 redo을 요구하는 결정은 시스템 고장으로부터 회복하기 위한 요구 사항의 견해에 의해 결정된다. 이러한 undo와 redo은

각각의 고유의 규칙을 가지고 있다.

Undo 규칙 : 변경된 데이타의 내용이 디스크 내의 데이타베이스로 옮겨지기 전에 이 데이타에 해당되는 로그 레코드가 먼저 안전한 기억장치의 로그에 기록되어야 하는 로그 우선 기록 프로토콜(write ahead log protocol)을 만족하여야 한다. 만약 현재 안정된 데이타베이스 안의 x의 영역에 x의 가장 최근 완료된 내용을 포함하고 있으면 이 내용은 완료되지 않았던 내용에 의해 안정된 데이타베이스 안에 변경되기 전에 안정된 기억장치 안에 저장되어야 한다.

Redo 규칙 : 한 트랜잭션이 완료하기 전에 이 트랜잭션의 각 데이타 요소에 대하여 변경된 내용은 안정된 기억장치(안정된 데이타베이스 또는 로그) 안에 있어야 한다.

Undo와 redo 규칙은 각 데이타 요소의 가장 최근에 완료된 내용이 항상 안정된 기억장치 안에서 이용될 수 있어야 하는 성질을 가지고 있다. 따라서 undo 규칙은 각 데이타 요소의 가장 최근에 완료된 내용이 완료되지 않았던 내용에 의하여 변경되기 전에 안정된 기억장치(즉, 로그)에 저장되도록 보장되어야 하고, redo 규칙은 각 데이타 요소의 값이 변경되는 순간에 안정된 기억장치 안에 있도록 보장되어야 한다.

가. Undo/Redo 알고리즘

이 알고리즘은 고장 회복을 위한 4가지 방법 중 가장 복잡하다. 그러나 다음과 같은 2 가지 장점을 가지고 있다. 첫째는 undo/redo를 사용하는 복귀 방법은 불필요한 *flush*가 자주 일어나는 것을 피할 수 있다. 이로 인해 I/O를 최소화할 수 있다. 이에 반해 no-redo의 경우는 일반적으로 트랜잭션의 변경된 요소의 모두가 안정된 데

이타 베이스에 있도록 보장되어야 하므로 보다 빈번하게 *flush*를 해야 한다. 둘째는 임의의 완료된 트랜잭션에 의해 마지막으로 쓰여진 변경된 내용을 교체하기 위해 *in-place updating*을 사용하는 것을 허용한다.

트랜잭션 T_i가 내용 v를 데이타 요소 x에 쓴다고 가정하면, 이 알고리즘에서는, 만약 x가 주기억장치 안에 없으면 x를 *fetch*하고, 로그와 x의 주기억장치 영역에 v를 기록한다. 이때 x에 대한 주기억장치 내용을 *flush* 하도록 요구하지는 않는다. 또 다른 데이타 요소를 *fetch*하기 위하여 x의 영역을 회수하고자 할때에는 x의 내용을 안정된 데이타 베이스로 *flush*한다. 만약 변경된 내용을 교체하고 T_i가 abort되거나 시스템 고장이 발생된 경우 undo가 요구된다. 그리고 만약 변경된 내용을 교체하기 전에 T_i가 종료되거나 또는 시스템 고장이 일어나면 redo가 요구된다.

나. Undo/No-redo 알고리즘

이 알고리즘은 주로 redo를 요구하지 않는 시스템에서 사용된다. 이 알고리즘을 성취하기 위해서는 알고리즘은 트랜잭션이 완료되기 전에 안정된 데이타베이스 안에 모든 트랜잭션의 변경 사항을 기록하여야 한다. 따라서 이 알고리즘은 매 트랜잭션이 완료될 때마다 버퍼 관리 프로그램에게 이 트랜잭션이 변경한 데이타 페이지를 디스크로 *flush*하도록 하여야 한다.

다. No-undo/Redo 알고리즘

이 알고리즘은 redo는 요구하지만 undo는 요구하지 않는 고장회복 방법이다. Undo를 하지 않기 위해서 안정된 데이타 베이스 안에 완료되지 않았던 트랜잭션의 변경된 내용들의 기록을 못하게 한다. 이러한 이유 때문에 데이타 요소가 변경될 때에 새로운 내용은 그 시점에서 주기억장치에 기록되지 않고, 꼭 트랜잭션이 완료된 후에 이루어진다. 결과적으로 새로운 내용은 교체되거나 *flush*되는 주기억장치 영역의 결과와

같이 안정된 데이타베이스 안에 기록될 때는 위의 내용은 완료된 트랜잭션의 내용이어야 한다. 따라서 결코 undo을 할 필요가 없다.

라. No-undo/No-redo 알고리즘

이 알고리즘은 undo와 redo 모두를 하지 않는 방법이다. Redo를 하지 않기 위하여 한 트랜잭션 T_i의 변경된 모든 내용들은 T_i가 완료되는 시점에서 안정된 데이타 베이스 안에 있어야 한다. 또 undo을 하지 않기 위해서는 T_i의 변경된 내용 모두가 T_i가 완료되기 전에 안정된 데이타 베이스 안에 있어서는 안된다. 그러므로 undo와 redo 모두를 제거하기 위해서는 T_i의 변경된 모든 내용들은 T_i의 완료 시점 때에 한 원자적 연산 내에서 안정된 데이타 베이스 안에 기록되어야 한다.

5.2.3 검사점을 이용한 시스템 고장으로부터의 회복

가. 검사점 방법들

시스템의 고장으로부터 시스템을 재개(restart)하기 위해서는 안전 기억장치에 기록되어 있는 모든 로그 레코드를 조사해야 한다. 그런데, 시스템에 고장이 발생할 때에 데이타베이스 내의 대부분의 데이타 항목들은 그 때까지 완료된 트랜잭션의 결과를 반영하고 있다. 따라서 시스템의 재개는 필요 이상의 많은 일을 하게 된다.

시스템의 고장이 회복될 때까지 어떤 일도 처리할 수 없으므로, 효율적인 시스템의 재개를 위한 방법이 필요하다. 이를 위한 방법으로 검사점(checkpointing) 기법이 사용된다. 검사점 기법은 고장으로부터 시스템을 재개하기 위해 요구되는 일의 양을 줄이기 위해서, 시스템이 정상적으로 동작할 때에 그때의 상태를 나타내는 여러가지 정보를 안전 기억장치에 기록하는 행위이다. 검사점은 다음 3 단계를 거쳐서 수행된다.

a. 로그화일에 시작-검사점 로그 레코드를 기록한다.

b. 로그화일에 데이타베이스에 모든 감사점 정보를 기록한다.

c. 로그화일에 종료-검사점 로그 레코드를 기록한다.

(1) 트랜잭션 위주의 검사점(transaction oriented checkpoint)

트랜잭션 위주의 검사점은 트랜잭션이 끝날 때마다 그 트랜잭션이 수정한 모든 페이지를 데이타베이스에 반영하여 국부적 재실행을 제거하는 방법이다. 즉, 검사점 이전에 끝난 트랜잭션은 재실행할 필요가 없다.

이 검사점을 이용한 시스템의 재실행은 시스템 고장시에 로그 정보를 이용하여 실행중인 트랜잭션들을 철회하면 된다.

트랜잭션 위주의 검사점의 주된 단점은 자주 참조되는 페이지들은 오랜 시간동안 버퍼에 남아 있으면서 트랜잭션에 의해 수정되어질 때마다 그 페이지를 데이타베이스에 반영하여야 한다. 어떤 페이지가 데이타베이스 버퍼에 오래 남아 있을수록 다른 트랜잭션에 의해 여러 번 수정될 가능성이 높다.

따라서 자주 참조되는 페이지가 많은 큰 데이타베이스 응용에서는 정상적인 처리를 하는 동안 부가 노력이 많이 발생하므로 트랜잭션 위주의 검사점은 이 응용에 적합하지 못하다.

(2) 트랜잭션-일치 검사점(transaction consistent checkpoint)

트랜잭션-일치 검사점은 데이타베이스를 수정한 모든 트랜잭션의 작업을 저장하는 점에 있어서 전역적이다. 트랜잭션-일치 검사점은 검사점 신호가 발생되어질 때 실행중인 트랜잭션은 계속 실행하여 완료하고 새로운 트랜잭션은 받아들이지 않는다.

실행중인 트랜잭션의 마지막 갱신이 이루어졌을 때 검사점이 생성된다. 종료-검사점 레코드가 성공적으로 로그화일에 쓰여지후 정상적인 트랜잭션의 실행이 다시 시작된다.

트랜잭션-일치 검사점은 국부적 재실행이 필요없는 시점을 설정한다. 즉, 최근의 검사점 이전에 발생한 모든 데이타 수정은 이미 데이타베이스에 반영되어 있으므로, 재실행 정보는 가장 최근의 종료-검사점 로그 레코드 이후의 로그만이 필요하다. 따라서 최근의 검사점 실행 이후로 완료되지 않은 트랜잭션들은 철회하고, 완료된 트랜잭션은 로그 정보를 이용하여 트랜잭션의 연산을 재실행하거나 갱신된 데이타의 값으로 데이타베이스에 반영하면 된다.

트랜잭션-일치 검사점 신호이후 새로운 트랜잭션을 허용하지 않고 버퍼가 큰 경우에는 검사점 비용이 높아지므로 작은 응용과 단일 사용자 시스템에서 유용하다.

(3) 연산-일치 검사법(action consistent checkpoint)

트랜잭션은 레코드 단위 레벨에서 데이타 조작언어(DML) 문장으로 처리되는 기본적인 연산의 나열로 생각할 수 있다. 연산-일치 검사점은 검사점 신호가 보내졌을 때 새로운 연산을 시작하지 않는다.

트랜잭션-일치 검사점 방법과 개념이 유사하지만 트랜잭션이 연산으로 세분화되었기 때문에 연산을 지연시키는 시간은 덜 걸린다. 버퍼가 큰 경우에는 여전히 검사점 비용이 높다.

(4) Fuzzy 검사점

검사점 수행시에 버퍼내에 수정된 페이지를 데이타베이스에 반영하지 않고 로그화일에만 그 정보를 기록한다. 이러한 검사점 방법은 데이타베이스는 현재 상태로 남겨둔 채로 로그화일에만 데이타의 변경을 반영하므로 'fuzzy'라는 용어를 사용한다.

이 방법의 변형된 형태로는 검사점 순간에 현재 버퍼에 있는 모든 수정된 페이지들은 로그화일에 쓰여진다. 또 다른 변형된 방법은 최근의 두번의 연속된 검사점사이에 수정된 페이지가 데이타베이스에 반영되지 않았다면 검사점 발생시에 이 페이지

는 데이타베이스에 반영된다[Hae 83].

나. 검사점을 고려한 시스템 회복

시스템 고장으로부터 위의 검사점을 고려한 회복과정은 모든 방법이 거의 유사하다. 따라서 위의 검사점 방법 중에서 가장 포괄적인 Fuzzy 검사점을 사용할 때의 회복과정을 설명한다.

시스템의 고장으로부터 시스템을 재실행할 때에는 다음 단계에 따라 회복한다. 먼저, 모든 버퍼 슬롯의 내용을 제거한다. 이는 시스템 고장시 주기억장치의 내용이 파괴되어 버퍼내의 내용을 신뢰할 수 없기 때문이다. 다음에는 로그를 후방으로 스캔(scan)해 가면서 완료되지 않거나 철회된 트랜잭션의 갱신을 철회하고, 이어서 전방으로 스캔해 가면서 완료된 트랜잭션의 갱신을 재수행한다.

(1) 후방 스캔

후방 스캔은 로그의 끝에서 부터 시작한다. 후방으로 스캔해 가면서 완료된 트랜잭션들과 완료되지 않은 트랜잭션들의 리스트를 각각 나타내는 CL과 AL을 생성한다. 로그 레코드[T_i, commit] 혹은 [T_i, abort]이면 해당되는 트랜잭션 T_i를 CL 혹은 AL 리스트에 첨가한다.

그리고, 레코드 RecID를 갱신한 트랜잭션 T_i에 대한 로그 레코드에 대해서는 아래의 단계에 따라 처리한다. 각 단계를 설명하기 전에, 앞에서 기술한 검사점 기법을 다시 생각 해 보자. 검사점 레코드 안에는 검사점 당시에 dirty 버퍼 슬롯에 있는 페이지에 대한 정보를 포함한다. 이를 이용하여 데이타베이스를 빨리 회복할 수 있다.

* 만약에 트랜잭션 T_i가 CL안에 있으면, 이 갱신 로그 레코드는 무시한다.
* 트랜잭션 T_i가 CL과 AL 어디에도 없으면, 트랜잭션 T_i를 AL에 첨가한다. 이는 시스템에 고장이 발생할 시점에 처리 중에 있는 트랜잭션이기 때문이

다.

* 트랜잭션 T_i가 AL 안에 있으면,

(a) 아래의 두가지 조건 중에 하나를 만족하면 이 갱신 로그 레코드는 무시해도 된다.

조건 1: 트랜잭션 T_i의 [T_i,abort] 로그 레코드가 마지막 검사점과 이 검사점 바로 이전의 검사점 사이에 있으면서, 마막 검사점 당시에 dirty 버퍼 슬롯의 어디에도 이 데이타 항목(즉 레코드 RecID)가 존재하지 않는 경우.

조건 2: T_i의 [T_i,abort] 로그 레코드가 마지막 검사점과 이 검사점 바로 이전의 검사점 사이에 있고 RecID가 마지막 검사점 당시의 임의의 dirty 버퍼 슬롯에 있으나, 이 슬롯 내에 있는 stable_LSN이 트랜잭션 T_i의 철회 로그 레코드의 LSN보다 큰 경우(stable_LSN은 이 페이지를 마지막으로 갱신한 연산의 로그 레코드에 대한 일련 번호이다).

(b) 위의 조건 중의 어느 것도 만족하지 않는 경우에는 아래의 일을 수행한다.

- 레코드 RecID의 페이지가 버퍼에 있지 안으면, 디스크 내의 데이타베이스에서 해당되는 페이지를 버퍼로 fetch해 온다. 다음에 갱신 로그 레코드에 기록된 레코드 RecID의 갱신되기 이전의 값을 가지고 버퍼의 내용을 변경한다. 그리고나서, 이 갱신 로그 레코드가 트랜잭션 T_i에 대한 최초의 로그 레코드라면 AL에서 트랜잭션 T_i를 제거한다.

위의 후방 스캔의 3번 단계에서 알 수 있듯이, 후방으로 스캔해 가다가 마지막 검사점 레코드에 이르면 검사점 당시의 dirty 슬롯에 대한 정보를 주기억장치에 보관

하고 후방 스캔을 계속한다. 다음에 마지막 검사점 바로 이전의 검사점 레코드에 이르면, 이 검사점 레코드에 저장된 그 당시 처리중인 트랜잭션 리스트(즉 active_list)를 조사하여 지금까지 유지하고 있는 AL이나 CL 어디에도 속하지 않는 트랜잭션은 AL리스트에 첨가한다. 이는 이 검사점 당시에 처리중이었으며 이후에 종료되지 않고 시스템에 고장이 발생할 당시에도 처리중에 있는 트랜잭션이기 때문이다. 마지막 바로 이전의 검사점 레코드를 처리하고 나서도 계속해서 후방으로 스캔해 가면서 AL 리스트에 있는 트랜잭션에 대한 갱신 로그 레코드는 위 알고리즘의 단계 3 에 따라 처리하고, 이 로그 레코드 이외의 모든 레코드는 무시한다.

위의 과정을 계속하다가 AL 리스트에 아무 것도 없을 때에 이 처리를 중단한다.

(2). 전방 스캔

위의 후방 스캔에 의한 처리가 끝나면 시스템에 고장이 발생할 당시의 처리중이거나 이전에 철회된, 즉 완료되지 않은 트랜잭션에 대한 철회가 모두 끝나게 된다. 이후에 하여야 할 일은 시스템이 고장 당시의 데이타베이스를 이전에 완료된 트랜잭션의 결과를 반영하는 상태로 회복하여야 한다. 이를 위하여, 마지막 바로 이전의 검사점 레코드에서 부터 시작하여 로그를 전방으로 스캔해 가면서 각 로그 레코드에 따라 아래의 일을 수행한다.

- 갱신 로그 레코드 이외의 로그 레코드는 무시한다.
- 갱신 로그 레코드 내의 트랜잭션 T_i가 CL에 있으면서 이 갱신 레코드를 포함한 페이지가 버퍼 내에 없으면, 디스크 내의 데이타베이스로부터 해당 블럭을 버퍼로 fetch해 온다. 그리고 나서 갱신 로그 레코드 내의 레코드 RecID의 갱신 후의 값을 버퍼 내의 슬롯에 반영한다. 갱신 로그 레코드 내의 트랜잭션 T_i가 CL에 있지 않으면 이 레코드는 무시한다.

후방 스캔 처리와 마찬가지로, 검사점 레코드의 내용을 이용하여 CL에 있는 트랜잭션에 대한 갱신 로그 레코드가 아래의 조건 중 하나를 만족하면, 이 레코드의 내용을 무시해 버릴 수 있다.

조건 1: 트랜잭션 T_i의 갱신 로그 레코드가 마지막 검사점과 이 검사점 바로 이전의 검사점 사이에 있으면서, 갱신 로그 레코드 내의 레코드 항목(즉 레코드 RecID)가 마지막 검사점 당시에 버퍼 내의 어느 dirty 슬롯에도 있지 않은 경우.

조건 2: 트랜잭션 T_i의 갱신 로그 레코드가 마지막 검사점과 바로 이전의 검사점 사이에 있고 갱신 로그 레코드 내의 레코드 항목(즉 레코드 RecID)가 마지막 검사점 당시에 버퍼 내의 어느 dirty 슬롯에 있으나, 이 슬롯 내의 stable_LSN이 갱신 로그 레코드 내의 LSN 보다 큰 경우.

검사점은 주기적으로 회복 관리 프로그램에 의하여 시작한다. 검사점을 수행하는 부가노력과 시스템의 재개시에 로그를 빨리 처리하는 장점 사이의 상반성을 고려하여 적절한 검사점 주기를 선정하여야 한다. 검사점 주기를 결정하는 여러가지 방법이 있으나, 본 연구에서는 검사점 이후 처리된 갱신 로그 레코드가 어느 한계값을 넘을 때에 검사점을 수행하도록 한다.

제 3 절 개발한 시스템에서의 회복 관리

개발한 시스템에서는 앞 절에서 언급한 여러가지 고장 중에서 단지 트랜잭션 고장으로부터의 회복만을 구현하였다. 트랜잭션이 처리되는 두가지 형태가 있다. 하나는 정상적인 트랜잭션의 완료이고, 다른 하나는 여러가지 원인으로 비정상적인 트랜잭션의 종료이다. 이에 대한 트랜잭션의 처리형태가 그림 5.1에 나타나 있다.

TBEGIN	TBEGIN
kass_insertrecord	kass_insertrecord
kass_writerecord	kass_writerecord
...	...
COMMIT	ABORT
a. 정상적인 완료	b. 비정상적인 종료정

그림 5.1 트랜잭션의 처리 형태

위의 그림 5.1의 b와 같은 트랜잭션은 ABORT 연산에서 트랜잭션의 철회를 실행한다. 트랜잭션의 철회는 트랜잭션이 실행되기 이전의 일관성 있는 상태로 데이타베이스를 회복시켜야 한다. 즉, 이런 경우에 시스템은 그 트랜잭션을 철회하여야 한다. 트랜잭션의 고장으로부터 회복하는 방법중에서 본 시스템에서는 로깅 방법을 사용하였다. 이는 공유성이 많은 대용량의 데이타베이스 체제에서는 로깅 기법이 좋은 성능을 나타낸다고 알려져 있으므로([Ber 83], [Reu 84]), 본 연구의 목표에 부합되기 때문이다.

로깅 방법은 기록하는 내용에 따라 물리적인 로깅(또는 데이타 값 로깅)과 논리적인 로깅(연산 로깅)으로 나누어진다. 물리적인 로깅은 데이타 조작언어 레벨 또는 레코드 레벨의 연산을 실행할 때에 변경되는 모든 데이타에 대하여(색인 트리도 포함), 변경되기 전의 값과 변경후의 값을 로그에 기록하는 방법이다. 반면에 논리적인 로깅은 데이타 조작언어 레벨 또는 레코드 레벨의 연산에 대하여 그연산 자체를 안전한 기억장치의 로그에 기록하는 방법이다. 그러나, 물리적인 로깅은 한 연산에 대하여 많은 양의 로그 레코드가 필요하고, 논리적인 로깅은 한 연산의 실행 시간이 길어져서 연산 실행중에 시스템 고장이 발생하는 경우에 일관된 데이타베이스 상태로 회복

하기가 어렵다. 실제로, 많은 시스템은 물리적 로깅 방법을 쓰거나 혼합된 방법을 사용하고 있다.

개발한 시스템의 저장구조 시스템이 잘 정의된 레코드 연산을 상위 레벨에 인터페이스로 제공함으로, 본 시스템은 로그 단위를 레코드로 하고 있다. 레코드 연산에 대해서는 레코드의 갱신되기 전의 값과 갱신후의 값 그리고 연산의 종류를 기록하는 변형된 논리적인 로깅 방법을 사용하였다. 또 비레코드(즉 트랜잭션) 연산에 대해서는 연산을 위한 정보 모두를 기록하였다.

5.3.1 로그 관리

개발한 시스템은 트랜잭션 고장만을 다루지만, 앞으로 시스템 고장을 다루기 용이한 로깅 방법을 구현하였다. 로깅 기법을 사용하게 되면 데이타베이스를 갱신하기 전에 그에 대한 로그 레코드를 로그에 기록하여야 어떠한 시점에서의 트랜잭션 고장으로부터도 회복이 가능하다. 로깅 기법에서 로그를 효율적으로 관리하기 위하여 로그의 단위를 어떻게 할 것인지가 문제가 된다. 본 연구에서는 로그 단위를 레코드로 한다.

로그 단위를 페이지로 하게 되면 어떤 트랜잭션이 한 페이지의 여러 레코드를 갱신하는 경우에, 각 레코드가 갱신될 때마다 그 페이지가 로그되어야 한다. 즉, 각 레코드가 갱신될 때마다 디스크의 입출력(I/O)이 요구되는 부가노력(overhead)이 필요하다. 그러나, 레코드 단위로 로그하는 경우에는 주기억장치에 로그 버퍼(buffer)를 두어 레코드에 대한 정보를 이 버퍼에 저장하였다가, 이 버퍼가 가득차게 되면 디스크로 옮겨 디스크 입출력 부가노력을 줄일 수 있다.

로그 버퍼를 이용하는 것은 완료되지 않은 트랜잭션이 갱신하는 레코드의 이전의 값이 안전 기억장치에 로그되기 전에, 이 갱신을 안전 데이타베이스에 반영할 수 있으므로 철회 규칙을 저해한다. 이러한 문제점을 해결하기 위해서 각 로그 레코드에

로그에 대한 일련 번호(log sequence number; LSN)를 부여하고, 버퍼 슬롯에 LSN을 포함하는 한 필드(field)를 첨가한다.

예를 들어, 연산 kass_writerecord(T_i,OpenFileNum,RecID,RecAdr,Len)의 처리는 먼저 이 연산에 대한 적절한 로그 정보를 로그 버퍼에 삽입하고 나서 버퍼 슬롯의 레코드 RecID의 내용을 갱신한다. 다음에 로그 레코드의 LSN을 버퍼 슬롯의 LSN에 기록한다.

버퍼의 한 슬롯의 내용이 디스크의 데이타베이스로 옮겨져야 한다면(flush out), 이 슬롯 내의 LSN과 같은 로그 레코드와 이전의 로그 레코드를 먼저 안전 기억장치로 옮겨야 한다. 즉, 레코드가 갱신될 때 갱신된 레코드의 내용을 디스크로 옮기기 전에 해당되는 로그 버퍼가 먼저 디스크로 저장되는 로그 우선 기록 프로토콜을 사용한다.

이러한 로깅 기법을 위하여 필요한 기억장치의 계층도가 그림 5.2에 나타나 있다.

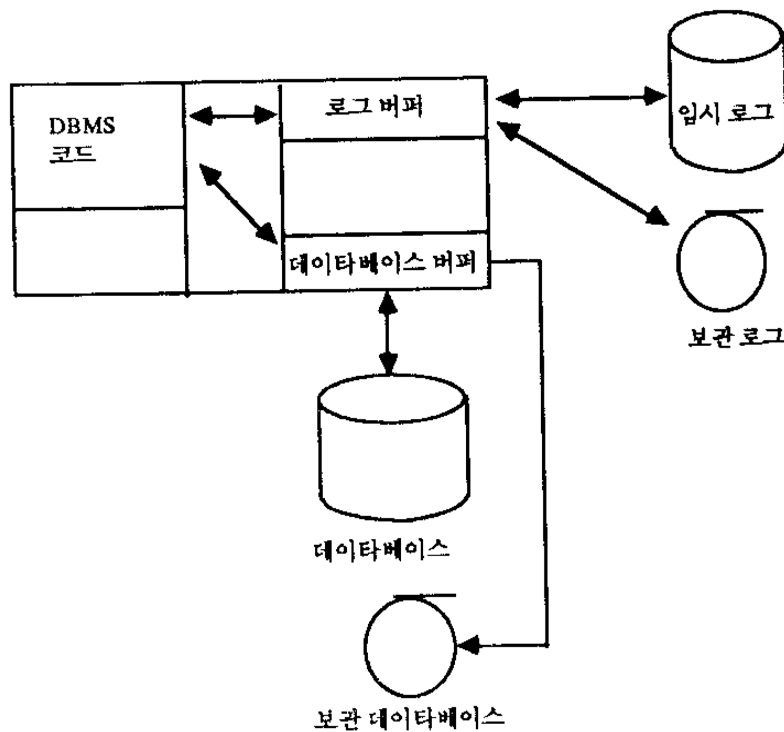


그림 5.2 회복을 위한 기억장치 계층도

그림 5.2에서 임시 로그(temporary log)와 보관 로그(archive log)는 로그 버퍼가 가득차거나 트랜잭션이 완료될 때에 기록되는 정보이다. 여기서 임시 로그는 디스크에 저장되며 트랜잭션 고장이나 시스템 고장시에 직접 액세스하여 고장으로부터 회복을 위한 것이다. 보관 로그는 자기 테이프에 저장하여 디스크 장치 고장으로부터 회복을 위함이다. 이와 같이 다른 두 가지 기억장치에 로그를 기록하는 것은 안전 기억장치를 구현하는 방법이다.

DRM은 그림 5.2의 로그 버퍼를 초기화하고, 스케쥴러나 질의 처리 프로그램으로부터 받은 각 연산에 대한 로그 레코드를 생성하여 이를 로그 버퍼에 첨가한다. 로그 버퍼를 초기화하고, 생성된 로그 레코드를 로그 버퍼에 첨가하는 모듈의 형태는 아래와 같다.

```
init_logbuffer(filenum)
int filenum
{
}

append_logbuffer(logrecord,reclen)
LOG_RECORD *logrecord; /* 생성한 로그 레코드 */
int reclen; /* 생성한 로그 레코드의 길이 */
{
}
```

마찬가지로, 데이타베이스도 디스크 내의 실제적인 사본과 테이프 내의 사본 두 가지로 둔다. 이때, 테이프 내의 사본은 시스템에 부하(load)가 거의 없는 시점에 덤프하여 보관한다. 앞에서 언급하였듯이, 본 시스템은 트랜잭션 고장을 위하여 로그

버퍼와 임시 로그만을 관리한다.

5.3.2 로그 레코드

개발한 시스템은 모든 연산에 대하여 연산 그 자체와 로그 관리를 위해 필요한 정보 그리고 레코드의 연산에 대한 갱신 전, 갱신 후의 값을 저장하는 논리적인 방법을 사용하였다. 시스템에서 사용한 로그 레코드의 정보는 다음과 같다.

- type : 연산의 형태
- filename : 연산에서 open한 화일
- tranid : 연산을 제기한 트랜잭션 식별자
- log_seq_no : 이 로그 레코드의 입력 번호
- prev_log_rec : 동일한 트랜잭션에 대한 바로 이전 로그 레코드의 포인터로, 저장된 페이지 번호와 페이지내의 슬롯 번호
- newrid : 생성되는 또는 연산이 실행되는 레코드의 고유번호
- nearrid : 연산될 레코드에 인접한 레코드 번호
- BV_len : 레코드의 갱신되기 전 값의 길이
- AV_len : 레코드의 갱신된 후 값의 길이
- fillfactor : 화일 또는 색인 생성시에 사용되는 fillfactor
- primary_flag : 연산의 종류에 따라 primaryflag, extentfillfactor, suffix등이 삽입되는 변수
- keynum : 색인 키의 번호(pagefactor을 저장하기도 함).
- data[] : 데이타의 갱신되기 전의 값과 갱신 후의 값을 저장한다.

5.3.3 트랜잭션 고장을 위한 준비 및 회복

회복 관리 프로그램은 트랜잭션이 시스템에 들어온 이후 실행되는 모든 연산에 대하여 트랜잭션 고장으로부터 회복에 필요한 정보를 로그 레코드에 삽입하여 로깅한다. 모든 연산에 대한 로그 레코드는 5.3.2 절의 형식에 따라 생성한다. 지면상 주된 일부의 연산에 대한 실행과정을 설명한다.

- TBEGIN(T_i) : 로그 버퍼에 로그 레코드 [begin, T_i]를 첨가하고, 스케쥴러에게 DONE 메세지를 전달한다.

- kass_writerecord(T_i,OpenFileNum,RecID,RecAdr,Len)
- kass_appendrecord(T_i,OpenFileNum,RecAdr,Len,NewRID)
- kass_insertrecord(T_i,OpenFileNum,RecAdr,Len,NearRID,NewRID)
- kass_deleterecord(T_i,OpenFileNum,RecID)

위 연산들에 대하여 아래의 단계에 따라 처리한다.

1. 트랜잭션 T_i가 실행중인 트랜잭션의 실행에 필요한 정보를 갖고 있는 ACT_QPIDS에 있지 않으면, 트랜잭션 T_i를 이 리스트에 첨가한다.
2. 레코드 RecID가 버퍼에 있지 않으면 이 RecID가 있는 블럭을 버퍼로 fetch해 온다.
3. 로그 레코드를 생성하여 로그 버퍼에 첨가한다.
4. 해당되는 연산을 실행하고, 해당되는 버퍼 슬롯의 dirty bit를 set한다. (원자적 행위)
5. 스케쥴러에게 DONE 메세지를 전달한다.

- kass_destroyfile(T_i,VolID,FileName)
- kass_droptree(T_i,VolID,FileName,IndexNo)
- kass_destroyhash(T_i,VolID,FileName,HashNo)

- kass_deletehash(T_i,OpenFileNum,Key,RecID)
- kass_destroylong(T_i,OpenFileNum,DirID)
- kass_destroy(T_i,VolID,FileName)

    1.- 3. 앞의 연산들과 동일.

    4. 만약에 이 연산들을 실행하고 나서 이 트랜잭션이 철회되면 이전의 상태로 회복이 불가능함으로, 이 연산들을 실행하지 않고 실행중인 트랜잭션 리스트인 ACT_QPIDS의 pend_list에 첨가한다(pend_list는 연쇄고리로 구성되어 있다).

    5. 스케쥴러에게 DONE 메세지를 전달한다.

- COMMIT(T_i) :

    1. 로그 레코드를 생성하여 로그 버퍼에 첨가하고, 로그 버퍼를 안전 기억장치에 기록한다.

    2. ACT_QPIDS의 pend_list에 있는 모든 연산을 실행한다.

    3. 스케쥴러에게 DONE 메세지를 전달한다.

    4. 트랜잭션 T_i를 ACT_QPIDS로부터 제거한다.

- ABORT(T_i) :

    1. 트랜잭션 T_i에 의해 실행된 각 연산에 대하여(ABORT 연산 바로전의 연산부터 철회하고, 로그 레코드의 prev_log_rec의 변수를 이용하여 전전 연산순으로)

        a. 이 레코드가 버퍼에 없으면 이 레코드를 포함한 블럭을 버퍼로 읽어 들인다.

        b. 로그를 참조하여 실행된 연산에 반대되는 연산을 실행한다(삽입이면 삭제 연산을). 특히 pend_list에 삽입된 연산이면 무시하고, prev_

log_rec를 이용하여 전 연산의 로그 레코드를 읽어 온다.

2. 트랜잭션 T_i에 대한 철회 로그 레코드를 로그 버퍼에 첨가한다.
3. 스케줄러에게 DONE 메세지를 전달한다.
4. 트랜잭션 T_i를 ACT_QPIDS로부터 제거한다. 이때에 pend_list에 있는 연산들을 삭제한다.

연산 중에서 kass_destroyfile과 같은 연산은 실행하고 나면, 이 연산의 철회가 불가능하다. 이와 같은 성질을 갖는 연산들은 회복 준비 단계에서 ACT_QPIDS의 pend_list에 유지하였다. 따라서 pend_list에 있는 연산들은 실행되지 않았으므로, 트랜잭션 철회시에 이러한 연산들은 무시하면 된다.

본 연구에서 구현한 시스템은 단지 트랜잭션 고장으로부터의 회복만을 다루고 있으나, 시스템 고장으로부터의 회복은 앞에서 생성한 로그 레코드를 그대로 이용할 수 있다. 시스템 고장으로부터의 회복을 위하여 검사점 방법과 시스템 재실행 연산을 구현하여야 할 것이다.

제 6 장 결론

본 연구에서는 UNIX를 기반으로 하며 관계형 데이타베이스 관리체제의 프로토타입을 설계하고 구현하였다. 질의어로는 ISO와 ANSI에서 표준 관계형 데이타베이스 언어로 정한 SQL을 이용하였다.

본 연구에서 개발된 관계형 데이타베이스 관리체제는 크게 질의 처리 시스템과 트랜잭션 처리 시스템으로 구성된다. 질의 처리시스템에서는 성능이 우수한 세 죠인 방법(중포 루프, 블록 중포, 병합 스캔)를 구현하였으며 휴리스틱 최적화 방법을 사용하였다. 트랜잭션 처리 시스템은 트랜잭션 스케쥴러와 데이타 회복 관리 프로그램으로 구성하였다. 동시성 제어 방법으로는 상충성 검사의 정도가 나쁘나 부가 노력이 적은 술어 로킹을 사용하였으며 회복 방법으로는 로깅을 사용하였다.

관계형 데이타베이스의 여러 기능 중 본 연구에서 개발한 프로토타입이 아직 제공하지 못하는 기능이 다수 있다. 먼저 일반 고급 언어와 연결하여 사용하는 기능이 없다. 또 폼과 보고서 작성기 등도 제공하지 않는다. SQL 중 보안 관련 명령도 제공하지 않으며 트랜잭션 고장에 대해서만 회복을 제공한다.

본 연구에서는 현재 개발한 데이타베이스 관리체제를 한글화하는 연구를 진행하고 있다. 이 연구는 한글 자료의 입출력뿐만 아니라 질의도 한글로 작성할 수 있게 해 준다. 한글코드로는 2 바이트 완성형을 선택하였고 현재 설계된 한글 질의어는 SQL과 같은 표현력을 가지나 한글의 구조에 적합하도록 설계되어 있다. 또한 개발한 데이타베이스 관리체제를 상품화할 수 있도록 확장하고 있다. 향후의 연구 과제로는 개발한 데이타베이스 관리 체제를 분산 데이타베이스 관리체제로 확장하는 연구와 다중 매체를 효율적으로 처리할 수 있도록 확장하는 연구를 계획하고 있다.

참 고 문 헌

[Bern 79] P. A. Bernstein, D. W. Shipman, and W. S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Transactions on Software Engineering*, SE-5, No. 3, May 1979, pp. 203-215.

[Bern 81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, Jun. 1981, pp. 185-221.

[Bern 83] P. A. Berstein, N. Goodman, and V. Hadzilacos, "Recovery Algorithms for Database Systems," Harvard University, TR-10-83, Mar. 1983.

[Bern 87] P. A. Berstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Blas 77] M. W. Blasgen and K. P. Eswaran, "Storage Access in Relational Database," *IBM System Journal*, Vol. 16, No. 4, 1977, pp. 363-377.

[Blas 81] M. W. Blasgen and et. al., "System R: An Architectual Overview," *IBM System Journal*, Vol. 20, No. 1, 1981.

[Ceri 84] S. Ceri and G. Pelagatti, *Distributed Databases, Principles and Systems*, McGraw-Hill Computer Science Series, 1984

[Ceri 85] S. Ceri and G. Gottlob, "Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries," *IEEE Transactions on Software Engeering*, SE-11, No. 4, April 1985.

[Crok 86] A. Croker and D. Maier, "A Dynamic Tree-Locking Protocol," *Proc. of Int. Conf. on Data Engineering*, 1986, pp. 49-56.

[Date 82] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, Vol. 1, 1982.

[Date 87] C. J. Date, *A Guide to the SQL Standard*, Addison-Wesley, 1987.

[Dina 84] Dina Bitton, D. J. Dewitt, and C. Turbyfill, "Benchmarking Database Systems A Systematic Approach," Computer Science Department, Univ. of Wisconsin-Madison, 1984, pp 8-19.

[Eswa 76] K. P. Eswaran, N. Gray, A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in A Database Systems," *Comm. of ACM*, Vol. 19, No. 11, Nov. 1976, pp. 624-633.

[Gray 81] J. N. Gray, et al., "The Recovery Manager of The System R Database Manager," *ACM Computing Surveys*, Vol. 13, No. 2, Jun. 1981, pp. 223-242.

[Gray 81a] J. N. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. of 7th Int. Conf. on Very Large Data Bases*, Nov. 1981, pp. 144-154.

[Haer 83] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, Dec. 1983, pp. 288-317.

[Hunt 79] H. B. Hunt III, and D. J. Rosenkrantz, "The Complexity of Testing Predicate Locks," *Proc. of Intl. Conf. on The Management of Data*, May 1979,

pp. 127-133.

[ISO 87] British Standards Institution, "British Standard Specification for Database Language SQL," 1987.

[John 78] Stephen C. Johnson, "YACC: Yet Another Compiler-Compiler," Bell Laboratories, 1978.

[Kim 87] M. J. Kim, et al., "Study on The Treatment of Korean Information in Database Management Systems," 과학 기술 86 특정 연구 결과 발표회 논문집, Jul. 1987.

[Kort 82] H. F. Korth, "Deadlock Freedom Using Edge Locks," *ACM Transactions on Database Systems*, Vol. 7, No. 4, Dec. 1982, pp. 632-652.

[Kung 81] H. T. Kung and J. T. Robinson, "An Optimistic Method for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, 1981, pp. 213-226.

[Lesk 75] M. E. Lesk, "Lex - A Lexical Analyzer Generator," Bell Laboratories, 1975.

[Lori 77] R. A. Lorie, "Physical Intergrity in A Large Segmented Database," *ACM Transactions on Database Systems*, Vol. 2, No. 1, Mar. 1977, pp. 91-104.

[Mack 86] L. F. Mackert and G. M. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *ACM SIGMOD*, 1986, pp. 84-95.

[Meec 87] D. J. Meechan and M. T. Ozsu, "Analysis of Access Path Selection in Database Systems," TR 87-7, Univ. of Alberta, Canada, May 1987.

[Papa 79] C. Papadimitriou, "The Serializability of Concurrent Database Updates," *Journal of ACM*, Vol. 26, No. 4, Oct. 1979, pp. 631-653.

[Papa 86] C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[Reut 83] A. Reuter and T. Haerder, "Principles of Transaction Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, Dec. 1983, pp. 287-317.

[Reut 84] A. Reuter, "Performance Analysis of Recovery Technique," *ACM Trans. on Database Systems*, Vol. 9, No. 4, Dec. 1984, pp. 526-559.

[Rodg 89] U. Rodgers, "UNIX Facilities and Constraints for Multiuser DBMS," *Database Programming & Design*, Oct. 1989.

[Schl 81] Schlageter, G. "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. of 7th Int. Conf. on Very Large Data Bases*, 1981. pp. 125-130.

[Shap 86] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Trans. on Database Systems*, Vol. 11, No. 3 Sep. 1986, pp. 239-264.

[Seli 79] P. G. Selinger, et. al., "Access Path Selection in a Relational Database

Management System," ACM SIGMOD 1979 *Int. Conf. on Management of Data*, May 30 - June 1, Boston, Massachusetts, pp. 23-34

[Silb 80] A. Silberschatz and Z. Kedem, "Consistency in Hierarchical Database Systems," *Journal of ACM*, Vol. 27, No. 1, 1980, pp. 72-80.

[Wong 76] E. Wong and K. Youssefi "Decomposition - A Strategy for Query Processing," *ACM trans. on Database Systems*, Vol. 1, No. 3, Jun. 1976, pp. 223-241.

부록 1. SQL 질의 구문

<< SCHEMA DEFINITION LANGUAGE >>

```
schema
    ::=    CREATE SCHEMA
                                AUTHORIZATION user
                                [ schema-element-list ]

schema-element
    ::=    base-table-def

base-table-def
    ::=    CREATE TABLE base-table ( base-table-element-commalist )

base-table-element
    ::=    column-def
                                | unique-constraint-def

column-def
    ::=    column data-type [ NOT NULL [ UNIQUE ] ]

unique-constraint-def
    ::=    UNIQUE ( column-commalist )
```

<< MANIPULATIVE STATEMENTS >>

```
manipulative-statement
    ::=    commit-statement
                                | delete-statement-positioned
                                | delete-statement-searched
                                | insert-statement
                                | rollback-statement
                                | select-statement
                                | update-statement-positioned
                                | update-statement-searched

commit-statement
    ::=    COMMIT WORK

delete-statement-positioned
    ::=    DELETE FROM table WHERE CURRENT OF cursor

delete-statement-searched
    ::=    DELETE FROM table [ where-clause ]

insert-statement
    ::=    INSERT INTO table [ ( column-commalist ) ]
```

```
insert-atom
    ::=     atom | NULL

rollback-statement
    ::=     ROLLBACK WORK

select-statement
    ::=     SELECT [ ALL | DISTINCT ] selection
                                        INTO target-commalist
                                        table-exp

update-statement-positioned
    ::=     UPDATE table SET assignment-commalist
                                        WHERE CURRENT OF cursor

assignment
    ::=     column = ( scalar-exp | NULL )

update-statement-searched
    ::=     UPDATE table SET assignment-commalist
                                        [ where-clause ]

<< QUERY EXPRESSIONS >>

query-exp
    ::=     query-term
                                        | query-exp UNION [ ALL ] query-term

query-term
    ::=     query-spec | ( query-exp )

query-spec
    ::=     SELECT [ ALL | DISTINCT ] selection table-exp

selection
    ::=     scalar-exp-commalist | *

table-expr
    ::=     from-clause
          [ where-clause ]
          [ group-by-clause ]
          [ having-clause ]

from-clause
    ::= FROM table-ref-commalist

table-ref
```

::= table [range-variable]

where-clause
::= WHERE search-condition

group-by-clause
::= GROUP BY column-ref-commalist

having-clause
::= HAVING search-condition

<< SEARCH CONDITIONS >>

search-condition
::= boolean-term
| search-condition OR boolean-term

boolean-term
::= [NOT] boolean-primary

boolean-primary
::= predicate | (search-condition)

predicate
::= comparison-predicate
| test-for-null
| like-predicate
| between-predicate
| in-predicate
| all-or-any-predicate
| existence-test

comparison-predicate
::= scalar-exp comparison (scalar-exp | subquery)

comparison
::= = | <> | < | > | <= | >=

between-predicate
::= scalar-exp [NOT] BETWEEN scalar-exp AND scalar-exp

like-predicate
::= column-ref [NOT] LIKE atom [ESCAPE atom]

test-for-null
::= column-ref IS [NOT] NULL

in-predicate

```
    ::=     scalar-exp [ NOT ] IN
          ( subquery | atom [, atom-commalist ] )

all-or-any-predicate
    ::=     scalar-exp comparison [ ALL | ANY | SOME ] subquery

existence-test
    ::=     EXISTS subquery

subquery
    ::=     ( SELECT [ ALL | DISTINCT ] selection table-exp )
```

<< SCALAR EXPRESSION >>

```
scalar-exp
    ::=     term
                                        | scalar-exp ( + | - ) term

term
    ::=     factor
                                        | term { * | / } factor

factor
    ::=     [ + | - ] primary

primary
    ::=     atom
                                        | column-ref
                                        | function-ref
                                        | ( scalar-exp )

atom
    ::=     parameter-ref
                                        | literal
                                        | USER

parameter-ref
    ::=     parameter [ [ INDICATOR ] parameter ]

function-ref
    ::=     COUNT(*)
                                        | distinct-function-ref
                                        | all-function-ref

distinct-function-ref
    ::=     { AVG | MAX | MIN | SUM | COUNT } ( DISTINCT column-ref )

all-function-ref
```

::= { AVG | MAX | MIN | SUM | COUNT } ([ALL] scalar-exp)

<< MISCELLANEOUS >>

table
::= base-table | view

user
::= authorization-identifier

column-ref
::= [column-quantifier .] column

column-quantifier
::= table | range-variable

target
::= parameter-ref

부록 2. SQL 어휘 분석용 정규 표현식

```
C       [A-Za-z]
D       [0-9]
E       [eE]?[-+]?{D}+
F       [A-Za-z_0-9]
G       [ \"\!\\                                  \$\%\  \
%%
[ \t\n] ;
(ALL|all)                                { return(S_ALL); }
(AND|and)                                { return(S_AND); }
(ANY|any)                                { return(S_ANY); }
(AUTHORIZATION|authorization)            { return(S_AUTHORIZATION); }
(AVG|avg)                                { return(S_AVG); }
(BETWEEN|between)                        { return(S_BETWEEN); }
(BTREE|btree)                            { return(S_BTREE); }
(BY|by)                                  { return(S_BY); }
(CHARACTER|character)                    { return(S_CHARACTER); }
(CHAR|char)                              { return(S_CHAR); }
(CLUSTER|cluster)                        { return(S_CLUSTER);
(COMMIT|commit)                          { return(S_COMMIT); }
(COUNT|count)                            { return(S_COUNT); }
(CREATE|create)                          { return(S_CREATE); }
(DATABASE|database)                      { return(S_DATABASE); }
(DECIMAL|decimal)                        { return(S_DECIMAL); }
(DEC|dec)                                { return(S_DECIMAL); }
(DELETE|delete)                          { return(S_DELETE); }
(DISTINCT|distinct)                      { return(S_DISTINCT); }
(DROP|drop)                              { return(S_DROP); }
(EXISTS|exists)                          { return(S_EXISTS); }
(EXTENT|extent)                          { return(S_EXTENT); }
(FLOAT|float)                            { return(S_FLOAT);
(FOR|for)                                { return(S_FOR); }
(FROM|from)                              { return(S_FROM); }
(GROUP|group)                            { return(S_GROUP); }
(HASH|hash)                              { return(S_HASH); }
(HAVING|having)                          { return(S_HAVING); }
(HELP|help)                              { return(S_HELP); }
(INDEX|index)                            { return(S_INDEX); }
(INDICATOR|indicator)                    { return(S_INDICATOR); }
(INSERT|insert)                          { return(S_INSERT); }
(INTEGER|integer)                        { return(S_INTEGER); }
(INT|int)                                { return(S_INT); }
(INTO|into)                              { return(S_INTO); }
(IN|in)                                  { return(S_IN); }
(IS|is)                                  { return(S_IS);
(MAX|max)                                { return(S_MAX); }
(MIN|min)                                { return(S_MIN);
```

```
(NOT|not)                      { return(S_NOT); }
(NULL|null)                    { return(S_NULL); }
(NUMERIC|numeric)              { return(S_NUMERIC); }
(OF|of)                        { return(S_OF); }
(ON|on)                        { return(S_ON); }
(OR|or)                        { return(S_OR); }
(PAGE|page)                    { return(S_PAGE); }
(PATH|path)                    { return(S_PATH);
(PRECISION|precision)          { return(S_PRECISION); }
(REAL|real)                    { return(S_REAL); }
(ROLLBACK|rollback)            { return(S_ROLLBACK); }
(SCHEMA|schema)                { return(S_SCHEMA); }
(SELECT|select)                { return(S_SELECT); }
(SET|set)                      { return(S_SET); }
(SHOW|show)                    { return(S_SHOW); }
(SMALLINT|smallint)            { return(S_SMALLINT); }
(SOME|some)                    { return(S_SOME); }
(START|start)                  { return(S_START); }
(STATISTICS|statistics)        { return(S_STATISTICS); }
(STOP|stop)                    { return(S_STOP); }
(SUM|sum)                      { return(S_SUM); }
(TABLE|table)                  { return(S_TABLE); }
(TO|to)                        { return(S_TO); }
(UNION|union)                  { return(S_UNION); }
(UNIQUE|unique)                { return(S_UNIQUE); }
(UPDATE|update)                { return(S_UPDATE);
(VALUES|values)                { return(S_VALUES);
(WHERE|where)                  { return(S_WHERE); }
(WITH|with)                    { return(S_WITH); }
(WORK|work)                    { return(S_WORK); }
\.                             { return(S_DOT); }
\(                             { return(S_LEFT_PAR); }
\,                             { return(S_COMMA); }
\)                             { return(S_RIGHT_PAR); }
\*                             { return(S_STAR); }
\+                             { return(S_PLUS); }
\-                             { return(S_MINUS); }
\=                             { return(S_EQUAL); }
\<                             { return(S_LESS); }
\>                             { return(S_GREATER); }
\<\>                           { return(S_NOT_EQUAL); }
\;                             { return(S_SEMICOLON); }
\/                             { return(S_SLASH); }
\"(\/|(({C})?({F})*)|\.|\~)+\"  { return(S_pathname); }
"'"({D}|{F}|\"|{G})+"'"        { return(S_character); }
{D}+"."{D}*({E})?              { return(S_numeric); }
"."{D}+({E})?                  { return(S_numeric); }
{D}+                           { return(S_unsigned_integer); }
```

```
{D}+{E}                                          { return(S_numeric); }
({C})?({F})*        { return(S_identifier); }
```

부록 3. YACC의 입력

```
sql_statement                       /* start symbol of grammar */
        : sql_element_LIST


sql_element_LIST
        : sql_element
        | sql_element_LIST  sql_element


sql_element
        : schema  S_SEMICOLON
        | commit_statement  S_SEMICOLON
        | delete_statement  S_SEMICOLON
        | insert_statement  S_SEMICOLON
        | rollback_statement  S_SEMICOLON
        | update_statement  S_SEMICOLON                 /* End SQL_statement */
        | query_specification  S_SEMICOLON
        | start_database  S_SEMICOLON
        | stop_database  S_SEMICOLON
        | drop_database  S_SEMICOLON
        | drop_index  S_SEMICOLON
        | drop_table  S_SEMICOLON
        | show_database  S_SEMICOLON
        | update_statistics_database  S_SEMICOLON
        | update_statistics_table  S_SEMICOLON
        | help_database  S_SEMICOLON
        | help_table  S_SEMICOLON
        | help_index  S_SEMICOLON
        ;

table_name
        : S_identifier
            authorization_identifier_OPTIONAL
        ;

authorization_identifier_OPTIONAL
        :               /* empty */
        | S_DOT  S_identifier
        ;

data_type
        : character_string_type
        | exact_numeric_type
        | approximate_numeric_type
        ;
```

```
character_string_type
            : character  length_braces_OPTIONAL
            ;

character
            : S_CHARACTER
            | S_CHAR
            ;

length_braces_OPTIONAL
            :               /* empty */
            | S_LEFT_PAR  length  S_RIGHT_PAR
            ;

exact_numeric_type
            : numeric  precision_scale_OPTIONAL
            | integer
            ;

numeric
            : S_DECIMAL
            | S_DEC
            | S_NUMERIC
            ;

integer
            : S_INTEGER
            | S_INT
            | S_SMALLINT
            ;

precision_scale_OPTIONAL
            :               /* empty */
            | S_LEFT_PAR  precision  comma_scale_OPTIONAL  S_RIGHT_PAR
            ;

comma_scale_OPTIONAL
            :               /* empty */
            | S_COMMA  scale
            ;

approximate_numeric_type
            : S_FLOAT  precision_braces_OPTIONAL
            | real
            ;

real
            : S_REAL
```

```
                ;

precision_braces_OPTIONAL
                :               /* empty */
                | S_LEFT_PAR  precision  S_RIGHT_PAR
                ;

length
                : S_unsigned_integer
                ;

precision
                : S_unsigned_integer
                ;

scale
                : S_unsigned_integer
                ;


value_specification
                : target_specification
                | literal
                ;

target_specification
                : S_identifier
                  indicator_OPTIONAL_variable
                ;

indicator_OPTIONAL_variable
                :               /* empty */
                | S_INDICATOR  S_identifier
                ;


column_specification
                : S_identifier
                | S_identifier
                  S_DOT  S_identifier
                ;


value_column_specification
                : column_specification
                | indicator_column_specification
                ;
```

```
indicator_column_specification
            : S_identifier
              indicator_column_OPTIONAL
            | literal
            ;

indicator_column_OPTIONAL
            : S_identifier
            | S_INDICATOR  S_identifier
            ;


set_function_specification
            : S_COUNT  S_LEFT_PAR  distinct_all_in_count  S_RIGHT_PAR
            | function_set  S_LEFT_PAR  all_or_distinct  S_RIGHT_PAR
            ;

distinct_all_in_count
            : S_STAR
            | S_DISTINCT  column_specification
            ;

all_or_distinct
            : S_DISTINCT  column_specification
            | all_OPTIONAL  value_expression
            ;

function_set
            : S_AVG
            | S_MIN
            | S_MAX
            | S_SUM
            ;

all_OPTIONAL
            :               /*  empty  */
            | S_ALL
            ;


value_expression
            : term
            | value_expression  plus_minus  term
            ;

plus_minus
            : S_PLUS
            | S_MINUS
```

```
        ;

term
        : factor
        | term  mult_div  factor
        ;

mult_div
        : S_STAR
        | S_SLASH
        ;

factor
        : primary
        | S_PLUS  primary
        | S_MINUS  primary
        ;

primary
        : value_column_specification
        | set_function_specification
        | S_LEFT_PAR  value_expression  S_RIGHT_PAR
        ;


predicate
        : comparison_predicate
        | between_predicate
        | in_predicate
        | like_predicate
        | null_predicate
        | quantified_predicate
        | exists_predicate
        ;


comparison_predicate
        : value_expression
          comp_op  value_expression_subquery
        ;

value_expression_subquery
        : value_expression
        | subquery
        ;

comp_op
        : S_EQUAL
```

```
                | S_NOT_EQUAL
                | S_LESS
                | S_GREATER
                | S_LESS S_EQUAL
                | S_GREATER S_EQUAL
                ;

between_predicate
                : value_expression not_OPTIONAL S_BETWEEN
                  value_expression S_AND value_expression
                ;

not_OPTIONAL
                :               /* empty */
                | S_NOT
                ;

in_predicate
                : value_expression
                  not_OPTIONAL S_IN subquery_in_value_list
                ;

subquery_in_value_list
                : subquery
                | S_LEFT_PAR in_value_list S_RIGHT_PAR
                ;

in_value_list
                : value_specification
                | in_value_list S_COMMA value_specification
                ;

like_predicate
                : value_expression not_OPTIONAL S_LIKE
                  pattern escape_character_OPTIONAL
                ;

escape_character_OPTIONAL
                :               /* empty */
                | S_ESCAPE escape_character
                ;

pattern
                : value_specification
                ;
```

```
escape_character
            : value_specification
            ;


null_predicate
            : column_specification S_IS not_OPTIONAL S_NULL
            ;


quantified_predicate
            : value_expression comp_op quantifier subquery
            ;

quantifier
            : S_ALL
            | S_SOME
            | S_ANY
            ;


exists_predicate
            : S_EXISTS subquery
            ;


search_condition
            : boolean_term
            | search_condition S_OR boolean_term
            ;

boolean_term
            : boolean_factor
            | boolean_term S_AND boolean_factor
            ;

boolean_factor
            : boolean_primary | S_NOT boolean_primary ;
boolean_primary
            : predicate
            | S_LEFT_PAR search_condition S_RIGHT_PAR
            ;


table_expression
            : from_clause
              where_clause_OPTIONAL
```

```
                group_by_clause_OPTIONAL
                having_clause_OPTIONAL
            ;

where_clause_OPTIONAL
            :           /* empty */
            | where_clause
            ;

group_by_clause_OPTIONAL
            :           /* empty */
            | group_by_clause
            ;

having_clause_OPTIONAL
            :           /* empty */
            | having_clause
            ;


from_clause
            : S_FROM  table_reference_LIST
            ;

table_reference_LIST
            : table_reference
            | table_reference_LIST   S_COMMA  table_reference
            ;

table_reference
            : table_name
              correlation_name_OPTIONAL
            ;

correlation_name_OPTIONAL
            :           /* empty */
            | S_identifier
            ;


where_clause
            : S_WHERE  search_condition
            ;


group_by_clause
            : S_GROUP  S_BY  column_specification_LIST
            ;
```

```
column_specification_LIST
            : column_specification
            | column_specification_LIST  S_COMMA  column_specification
            ;


having_clause
            : S_HAVING  search_condition
            ;


subquery
            : S_LEFT_PAR  S_SELECT  all_distinct
               result_specification  table_expression  S_RIGHT_PAR
            ;

all_distinct
            :              /*  empty  */
            | S_ALL
            | S_DISTINCT
            ;

result_specification
            : value_expression
            | S_STAR
            ;


query_specification
            : S_SELECT
               all_distinct  select_list  table_expression
            ;

select_list
            : value_expression_LIST
            | S_STAR
            ;

value_expression_LIST
            : value_expression
            | value_expression_LIST  S_COMMA  value_expression
            ;


schema
            : S_CREATE  S_SCHEMA  schema_authorization_clause
               schema_element_LIST
```

```
            ;

schema_element_LIST
            :               /* empty */
            | schema_element_LIST schema_element
            ;

schema_authorization_clause
            : S_AUTHORIZATION S_identifier
            ;

schema_element
            : db_definition
            | table_definition
            | index_definition
            ;


table_definition
            : S_CREATE S_TABLE table_name
              S_LEFT_PAR table_element_LIST S_RIGHT_PAR
              S_PAGE S_EQUAL S_unsigned_integer
            ;

table_element_LIST
            : table_element
            | table_element_LIST S_COMMA table_element
            ;

table_element
            : column_definition
            | unique_constraint_definition
            ;


column_definition
            : S_identifier data_type not_null_unique_OPTIONAL
            ;

not_null_unique_OPTIONAL
            :               /* empty */
            | S_NOT S_NULL unique_OPTIONAL
            ;

unique_OPTIONAL
            :               /* empty */
            | S_UNIQUE
            ;
```

```
unique_constraint_definition
            : S_UNIQUE  S_LEFT_PAR  column_list  S_RIGHT_PAR
            ;


commit_statement
            : S_COMMIT  S_WORK
            ;

delete_statement
            : S_DELETE  S_FROM  table_name  positioned_searched
            ;

positioned_searched
            : positioned
            | searched
            ;

target_specification_list
            : target_specification
            | target_specification_list   S_COMMA   target_specification
            ;


insert_statement
            : S_INSERT  S_INTO  table_name  insert_column_list_OPTIONAL
              values_query_specification
            ;

values_query_specification
            : S_VALUES  S_LEFT_PAR  insert_value_list  S_RIGHT_PAR
            | query_specification
            ;

insert_column_list_OPTIONAL
            :            /*  empty  */
            | S_LEFT_PAR  column_list  S_RIGHT_PAR
            ;

column_list
            : S_identifier
            | column_list  S_COMMA  S_identifier
            ;

insert_value_list
            : insert_value
```

```
               | insert_value_list  S_COMMA  insert_value
               ;

insert_value
               : value_specification
               | S_NULL
               ;


rollback_statement
               : S_ROLLBACK  S_WORK
               ;


update_statement
               : S_UPDATE  table_name
                 S_SET  set_clause_LIST  positioned_searched
               ;


positioned
               : S_WHERE  S_CURRENT  S_OF  S_identifier
               ;

set_clause_LIST
               : set_clause
               | set_clause_LIST  S_COMMA  set_clause
               ;

set_clause
               : object_column_positioned  S_EQUAL  value_expression_null
               ;

value_expression_null
               : value_expression
               | S_NULL
               ;

object_column_positioned
               : S_identifier
               ;


searched
               :               /*  empty  */
               | where_clause
               ;
```

```
literal
        : S_character
        | S_numeric
        | S_unsigned_integer
        ;

start_database
        : S_START  S_DATABASE  S_identifier
        ;

stop_database
        : S_STOP  S_DATABASE  S_identifier
        ;

drop_database
        : S_DROP  S_DATABASE  S_identifier
        ;

drop_table
        : S_DROP  S_TABLE  table_name
        ;

drop_index
        : S_DROP  S_INDEX  S_identifier
        ;

show_database
        : S_SHOW  S_DATABASE  S_identifier
        ;

db_definition
        : S_CREATE  S_DATABASE  S_identifier
          S_PATH  S_EQUAL  S_pathname
          S_PAGE  S_EQUAL  S_unsigned_integer
          S_EXTENT  S_EQUAL  S_unsigned_integer
        ;

index_definition
        : S_CREATE  hash_btree  S_INDEX  S_identifier
          S_ON  table_name
          S_LEFT_PAR  column_list  S_RIGHT_PAR
          unique_OPTIONAL  cluster_OPTIONAL
        ;

hash_btree
        : S_HASH
        | S_BTREE
        ;
```

```
cluster_OPTIONAL
            :            /* empty */
            | S_CLUSTER
            ;

update_statistics_database
            :  S_UPDATE  S_STATISTICS  S_DATABASE  S_identifier
            ;

update_statistics_table
            :  S_UPDATE  S_STATISTICS  S_TABLE  table_name
            ;

help_database
            : S_HELP  S_DATABASE  S_identifier
            ;

help_table
            : S_HELP  S_TABLE  table_name
            ;

help_index
            : S_HELP  S_INDEX  S_identifier
            ;
```

주 의

1. 이 보고서는 과학기술처에서 시행한 특정연구개발사업의 연구보고서이다.

2. 이 연구개발내용을 대외적으로 발표할 때에는 반드시 과학기술처에서 시행한 특정연구개발사업의 연구결과임을 밝혀야 한다.