고립화 수준을 검증하기 위한 트랜잭션의 시험

Testing Transactions based on Verification of Isolation Levels

홍석희
경성대학교 컴포터정보학부

Seok-Hee Hong(shong@ks.ac.kr)

요약

데이터베이스 응용 프로그램들은 DBMS가 관리하는 데이터베이스를 동시에 접근하기 때문에 동시성 문제가 발생하며 이를 해결하기 위해서 대부분의 상업용 DBMS들은 고립화 수준을 설정할 수 있게 한다. 데이터베이스 응용 프로그램을 구성하는 트랜잭션들의 고립화 수준에 따라서 데이터베이스의 일관성이 위반될 수도 있기 때문에 트랜잭션의 고립화 수준의 검증은 중요하다. 본 논문은 트랜잭션의 고립화 수준 설정 오류를 검증하기 위한 시험 도구를 제안하고 프로토타입을 구현한다. 시험 도구는 ESQL/C 프로그램을 분석하여 고립화 수준을 검증할 수 있는 코드를 추가하여 시험 트랜잭션을 실행시키고 고립화 수준의 오류를 검출할 수 있게 한다.

■ 중심어 : | 트랜잭션 | 고립화 수준 | 소프트웨어 시험 | SQL |

Abstract

Concurrency and synchronization problems are often caused by database applications concurrently accessing databases managed by DBMS. Most commercial DBMSs support isolation levels to resolve these problems. Verification of isolation levels are most important because consistency and integrity constraints of the database can be violated according to isolation levels of transactions that consists of database applications. We propose a test tool set to verify and reveal faulty settings of isolation levels and implement a prototype of the test tool set. The proposed tool set analyzes the SQL statements of ESQL/C programs, attaches the test codes to verify isolation levels, runs the test transactions and detects errors.

■ keyword : | Transaction | Isolation Level | Software Test | SQL |

I. 서 론

데이터베이스 관리 시스템(DBMS:Database Management Systems)은 특정한 목적을 위해서 서로 연관되어 있는 데이터의 집합체인 데이터베이스를 효율적이고 안전하게 운영하고 관리하는 기능을 한다. 최근 대부분의 정보화 시스템은 필수적으로 대용량 데이터베이스를 기반으로 많은 작업을 수행한다. 효율적으로 업무를 처리하기 위해서 동시에 많은 작업이 데이터베이스를 대상으로 검색과 수정을 하게 된다. 이와 같은 경우 신속하고 효율적으로 데이터베이스를 사용하는

* 본 논문은 2007학년도 경성대학교 학술연구비지원에 의하여 연구되었습니다.

접수번호 : #080506-004
접수일자 : 2008년 05월 06일
심사완료일 : 2008년 06월 09일
교신저자 : 홍석희, e-mail : shong@ks.ac.kr

것 뿐 아니라 검색된 데이터가 정확하고 수정된 데이터가 일관성(consistency)을 유지하도록 DBMS가 보장해야 한다.

상업용 DBMS들은 데이터베이스 응용 프로그램이 데이터베이스의 일관성과 무결성(integrity)을 유지할 수 있는 4단계의 고립화 수준(isolation Level)을 제공한다. 낮은 고립화 수준으로 설정된 트랜잭션은 정확하지 않은 데이터를 사용하게 되어 데이터베이스 응용 프로그램이 일관성이 결여된 실행 결과를 생성하지만 짧은 응답시간을 가진다. 높은 고립화 수준으로 설정된 트랜잭션은 항상 일관성을 만족하는 데이터만 사용하기 때문에 데이터베이스 응용 프로그램의 실행 결과를 신뢰할 수 있지만 응답시간이 길어지게 된다. 프로그램 개발자는 작업의 특성에 따라서 효율성과 일관성을 고려하여 적절한 고립화 수준을 선택해야 한다.

과거 수십 년 동안 소프트웨어 시스템의 동작이 요구조건에 부합하고 오류가 없는지를 시험하기 위한 많은 기법과 도구들이 개발되고 사용되어왔다. 이와 같은 소프트웨어 시험 도구를 통해서 복잡한 응용 프로그램의 개발 기간이 단축될 수 있었다[1]. 최근 데이터베이스를 기반으로 하는 많은 응용 프로그램들이 개발되고 있지만 기존의 소프트웨어 시험 기법과 도구를 활용하기가 쉽지 않다. 소프트웨어 공학 분야에서 연구 및 개발된 많은 소프트웨어 시험 기법들은 C 프로그래밍 언어와 같은 전통적인 절차형(imperative) 프로그래밍 언어로 작성된 응용 프로그램을 대상으로 한다. 따라서 선언형(declarative) 프로그래밍 언어인 SQL 문장들을 포함하는 데이터베이스 응용 프로그램을 시험하기 위해서 기존의 소프트웨어 시험 기법을 직접적으로 적용하기는 어렵다[2].

다양한 데이터베이스 언어와 프로그래밍 기법으로 개발되는 데이터베이스 응용 프로그램들은 데이터 저장소로 오라클이나 MS SQL 서버와 같은 상업용 DBMS가 관리하는 데이터베이스를 활용한다. 데이터베이스 응용 프로그램을 시험하기 위해서는 데이터베이스 상의 데이터의 입출력을 고려하여 소프트웨어 시험을 해야 한다. 따라서 기존의 소프트웨어 시험 기법과 다른 접근 방법이 고안되어야 한다. 본 논문에서는 데이터 일관성을 보장하기 위해서 고립화 수준을 설정해야 하는 데이터베이스 응용 프로그램의 오류를 시험하기 위한 자동화 도구를 제안한다.

본 논문의 구성은 다음과 같다. 2장에서 기존에 연구된 데이터베이스 응용 프로그램의 시험기법들을 논하고, 3장에서 고립화 수준 설정 오류로 인해서 발생하는 문제에 대해서 기술하고 4장에서 고립화 수준을 검증하기 위한 시험 기법을 고안 하고 시험 도구의 구현에 대해서 알아본다. 마지막으로 5장에서 결론과 향후 연구계획을 제시한다.

II. 관련 연구

전통적인 선언형 프로그래밍 언어로 작성된 프로그램을 위주로 소프트웨어 시험에 대한 많은 연구가 이루어진데 비하여 데이터베이스 응용 프로그램을 대상으로 하는 소프트웨어 시험 기법에 대한 연구는 많지 않다. 데이터베이스 응용 프로그램의 시험에 대한 대부분의 연구는 무결성 제약조건을 만족하는 데이터의 생성, white box 시험 기법에 기반을 둔 SQL 문장의 시험 등과 같은 분야에 집중되었다. 그러나 트랜잭션의 동기화 문제나 고립화 수준과 같은 동시성 기능에 대한 시험 기법의 연구는 미흡한 상황이다.

데이터베이스 응용 프로그램을 시험하기 위한 자동화된 시험도구로 AGENDA[2]는 많은 연구 성과를 보여주었다. AGENDA는 데이터베이스 응용 프로그램의 SQL 문장을 분석하여 시험에 필요한 데이터를 최소한의 크기로 생성하기 위한 기능을 제공한다. 또한 자동으로 생성한 데이터를 대상으로 SQL 문장을 시험하고 검증하여 시험자에게 오류 검출을 할 수 있게 도와준다. 데이터베이스의 무결성 제약조건을 쉽게 정의하고 트랜잭션의 SQL 문장들이 제약조건을 만족하는지 검증하는 기능을 AGENDA에 추가한 연구가 있었다[3]. 제약조건을 만족하는 시험 데이터를 자동으로 생성한 후 트랜잭션으로부터 추출한 SQL 문장들이 무결성 제약조건을 만족하는지 검증한다. 또한 [4]의 연구에서는 AGENDA에 트랜잭션 프로그래밍 오류로 인해 동시성

문제가 발생하는 문제를 검출하고 오류를 발생시킨 트랜잭션들을 부분적으로 재수행할 수 있는 기법을 연구하였다. 동시성 문제를 발생시킬 수 있는 두 트랜잭션들 사이의 오류를 검출하기 위해 데이터 충돌을 발생시키는 SQL 문장들을 분석하여 부분적인 재수행을 통한 시험 기법을 연구하였다. [4]의 연구는 SERIALIZABLE의 고립화 수준만을 고려하여 COMMIT과 ROLL BACK 문장에 대한 오류에 집중하였다.

소프트웨어 시험기법 중에서 오류를 수정한 프로그램이 새로운 오류를 유발시키지 않음을 검증하는 회귀시험(regression testing)을 트랜잭션 시험에 적용한 연구가 있었다[5][6]. 이 연구에서는 일련의 트랜잭션들을 실행하며 오류를 검증하는 과정에서 회귀시험의 비용을 최소화하기 위한 다양한 시험 기법들을 고안하였다.

III. 트랜잭션의 고립화 수준 설정 오류

1. ACID 성질

데이터베이스 응용 프로그램은 DBMS 환경 내에서 작업의 단위인 트랜잭션의 형태로 실행된다. 운영체제의 작업의 단위인 프로세스와 유사한 개념이지만 공유되는 데이터베이스를 동시에 많은 트랜잭션들이 접근할 수 있기 때문에 여러 가지 제약조건을 만족시켜야 한다. 이와 같은 제약을 데이터베이스 분야에서는 ACID 성질이라 하며 원자성(acid), 일관성(consistency), 고립성(isolation), 지속성(durability) 등의 네 가지 성질을 만족해야 한다.

다수의 트랜잭션들이 하나의 데이터베이스를 동시에 접근할 수 있게 하기 때문에 트랜잭션의 실행이 정확하지 않은 결과를 보여주거나 데이터베이스의 일관성이 위반 될 수도 있다. 각 트랜잭션의 실행이 일관성을 가지게 하려면 네 가지 ACID 성질을 모두 만족해야 한다. 그러나 네 가지 ACID 성질을 모두 만족하는 트랜잭션은 DBMS의 스케줄링 기법에 의해서 다른 트랜잭션을 기다리게 하는 대기시간이 길어져서 결과적으로 응답시간이 길어지게 된다. 그러나 일부 ACID 성질을 만족시키지 않는다면 상대적으로 트랜잭션의 응답시간이 짧아 질 수도 있다. 이와 같이 트랜잭션의 ACID 성질을 조절할 수 있도록 DBMS는 고립화 수준(isolation levels)을 설정할 수 있게 한다. 고립화 수준은 ANSI SQL-92 표준안에서 네 가지로 제안하고 있으며 대부분의 상업용 DBMS가 ANSI SQL-92 고립화 수준을 준수하고 있다[7][8].

2. 고립화 수준(isolation levels)

데이터베이스 응용 프로그램 개발자는 작업의 요구조건에 따라서 다음 네 가지 고립화 수준 중 하나를 선택해야 한다.

[Level 0] READ UNCOMMITTED : 실행중인 트랜잭션이 생성한 데이터를 사용한다. 데이터를 생성한 트랜잭션이 철회하는 경우 일관성이 위반될 수 있다.

[Level 1] READ COMMITTED : 종료된 트랜잭션이 생성한 데이터만을 사용하도록 보장한다.

[Level 2] REPEATABLE READ : 첫 번째 읽은 데이터는 그 이후에 다시 읽더라도 동일한 값으로 읽혀져야 함을 보장한다. 따라서 일단 읽은 데이터는 다른 트랜잭션이 삭제하거나 수정할 수 없게 한다.

[Level 3] SERIALIZABLE : 완전한 데이터 일관성을 보장한다.

고립화 수준이 낮을 수 록 동시에 실행되는 트랜잭션의 수를 의미하는 동시성 정도(degree of concurre ncy)가 높아진다. 따라서 트랜잭션들의 평균 응답시간이 향상된다. 그러나 트랜잭션이 사용하는 데이터의 일관성이 보장되지 않기 때문에 정확한 작업결과를 요구하는 트랜잭션에는 적합하지 않다. 반대로 고립화 수준이 높을 수 록 트랜잭션의 동시성 정도는 악화되지만 일관성을 보장받을 수 있게 된다. 고립화 수준의 정의를 위반하는 다음과 같은 세 가지 금지 상황을 현상(phenomena)이라고 한다[9].

[P0] 손상가능 읽기(dirty read) : 종료되지 않은 트랜잭션이 생성한 데이터를 읽은 것을 손상가능 읽기라고 한다. 고립화 수준 0에서 허용되는 일관성을 위반하는 연산에 해당한다. 고립화 수준 1은 손상가능 읽기를 허용하지 않도록 정의된다.

[P1] 비반복가능 읽기(non-repeatable read) : 첫 번

째로 읽은 데이터가 이후에 변경되는 경우에 해당하는 연산으로 반복가능 읽기가 위반되는 경우이다. 수정된 두 번째 읽은 데이터가 종료된 트랜잭션이 생성한 데이터라면 고립화 수준 1을 만족시킨 것이다.

[P2] 유령튜플(Phantom Tuple) : 테이블의 튜플을 여러 번 스캔하는 경우 두 번째 스캔 전에 새로운 튜플을 다른 트랜잭션이 삽입하는 경우 이 튜플을 유령 튜플이라 한다. 유령 튜플이 첫 번째 스캔 이후에 삽입되었기 때문에 반복가능 읽기를 위반한 연산이 아니므로 고립화 수준 2를 만족시킨다.

위 세 가지 현상들은 트랜잭션의 고립화 수준이 프로그램의 요구조건에 따라서 설정되었는지를 검증하기 위한 도구로 활용될 수 있다.

3. 트랜잭션 설계 오류

본 절에서는 데이터베이스 응용 프로그램 개발자가 고립화 수준을 잘 못 설정하는 경우 어떤 현상이 발생하고 데이터베이스 일관성을 어떻게 위반하는지를 예를 통해 보여준다. 데이터베이스 응용 프로그램을 개발하는 많은 방법론이 있지만 본 논문에서는 ESQL/C 방식으로 트랜잭션을 작성한다고 가정한다. ESQL/C 방식은 기본적으로 C 프로그래밍 언어로 데이터베이스 응용 프로그램을 작성하지만 데이터베이스에 대한 연산은 SQL 문장을 사용하여 실행한다.

표 1. 은행 데이터베이스 스키마

테이블명	애트리뷰트	의미
Account	No	계좌번호
	Balance	잔액
	Owner_No	계좌주의 주민등록번호

[표 1]과 같은 테이블 스키마를 가지는 은행업무 데이터베이스를 대상으로 트랜잭션 설계 오류의 예를 보이고자 한다. Account 테이블은 은행 계좌에 대한 정보를 표현하며 계좌번호(No), 잔액(Balance), 계좌주 번호(Owner_No) 등의 애트리뷰트를 가진다.

[그림 1]은 계좌 a1에서 a2로 amount 만큼의 액수를 이체하는 트랜잭션을 보여준다. 계좌 a1의 잔액이 amount 이상인지를 확인한 후 계좌 a1에서 amount 만큼의 잔액을 빼고 계좌 a2에 amount1 만큼의 잔액을 더한다. [그림 2]는 계좌 no에서 amount2 만큼의 액수를 입금하는 트랜잭션이다. [그림 3]은 계좌 no에서 amount2 만큼의 액수를 출금하는 트랜잭션이다. 이체 트랜잭션의 고립화 수준은 READ UNCOMMITTED로 설정되었고 인출 트랜잭션과 출금 트랜잭션은 고립화 수준을 설정하지 않았기 때문에 기본 값인 SERIALIZABLE로 설정되었다.

```
void Transfer() {
BEGIN DECLARE SECTION
  int a1, a2, amount1, balance;
END DECLARE SECTION
EXEC SQL SET TRANSACTION ISOLATION
 LEVEL READ UNCOMMITTED;
EXEC SQL SELECT Balance INTO :balance
  FROM Account WHERE No = :a1;
if(balance >= amount1) {
  EXEC SQL UPDATE Account
    SET Balance = Balance - :amount1
    WHERE No = :a1;
  EXEC SQL UPDATE Account
    SET Balance = Balance + :amount1
    WHERE No = :a2;
}
EXEC SQL COMMIT;
```

그림 1. 이체 트랜잭션

```
void Deposit() {
BEGIN DECLARE SECTION
  int no, amount2;
END DECLARE SECTION
EXEC SQL UPDATE Account
  SET Balance = Balance + :amount2
  WHERE No = :no;
...
EXEC SQL ROLLBACK;
}
```

그림 2. 입금 트랜잭션

호스트 변수 a1과 no의 값을 1234, a2를 4567, amount1을 100, amount2를 50이라 하자. 또한 트랜잭션 실행 전 Account 테이블의 계좌 a1과 a2의 잔액을 50이라 하자. 이체 트랜잭션과 입금 트랜잭션이 다음과 같은 시나리오로 실행된다고 가정하자.

① 입금 트랜잭션 : UPDATE 문장 실행

② 이체 트랜잭션 : SQL 문장 실행

③ 이체 트랜잭션 : 첫 번째 UPDATE 문장 실행
④ 입금 트랜잭션 : ROLLBACK 문장 실행
⑤ 이체 트랜잭션 : 두 번째 UPDATE 문장 실행
⑥ 이체 트랜잭션 : COMMIT 문장 실행

①번에 의해서 계좌 a1의 잔액은 100이 되고 ②번에 의해서 이체 트랜잭션은 계좌 a1의 잔액이 이체가 가능한 액수임을 확인한다. ③번에서 계좌 a1의 잔액은 0로 갱신된다. ④번에서 입금 트랜잭션이 ROLLBACK이 되어 ①번에서 입금한 액수는 50으로 복구된다. 이어서 ⑤번과 ⑥번에 의해서 이체 작업이 완료된다. 그러나 이체 트랜잭션이 이체한 금액 중 실제로 50은 존재하지 않는 액수로 데이터베이스의 일관성이 위반된 결과가 된다. 이와 같은 결과가 된 이유는 이체 트랜잭션의 고립화 수준이 READ UNCOMMITTED로 잘 못 설정되었기 때문이다. ②번의 SQL 문장이 읽은 balance는 손상가능 데이터(dirty data)로써 이후에 변경될 수도 있는 불완전한 값이다. 따라서 이체 트랜잭션의 고립화 수준 설정 오류가 수정되지 못한다면 이로 인해 데이터베이스의 일관성이 위반되어 심각한 문제가 발생할 수 있게 된다.

```
void Deposit() {
BEGIN DECLARE SECTION
   int no, amount2;
END DECLARE SECTION
EXEC SQL UPDATE Account
    SET Balance = Balance - :amount2
    WHERE No = :no;
...
EXEC SQL COMMIT;
}
```

그림 3. 출금 트랜잭션

이체 트랜잭션과 출금 트랜잭션이 다음과 같은 두 번째 시나리오로 실행된다고 하자. 호스트 변수의 값은 첫 번째 시나리오와 동일하고 계좌 a1의 잔액은 100으로 가정한다. 또한 이체 트랜잭션의 고립화 수준을 READ COMMITTED로 설정했다고 하자.

① 이체 트랜잭션 : SQL 문장 실행
② 입금 트랜잭션 : UPDATE 문장 실행
③ 입금 트랜잭션 : COMMIT 문장 실행
④ 이체 트랜잭션 : 첫 번째 UPDATE 문장 실행
⑤ 이체 트랜잭션 : 두 번째 UPDATE 문장 실행
⑥ 이체 트랜잭션 : COMMIT 문장 실행

①번에 의해서 계좌 a1의 balance 값인 100을 읽은 후 ②번에서 출금 트랜잭션이 계좌 a1의 balance 값을 50으로 감소한다. 이어 ③번에 의해서 출금 트랜잭션은 정상적으로 종료한다. 이어서 ④번과 ⑤번의 UPDATE 문장이 실행되면 계좌 a1과 a2의 잔액은 각각 -50과 150이 되고 ⑥번에서 실행을 종료한다.

고립화 수준이 READ COMMITTED이기 때문에 ①번의 balance 값은 정확한 데이터이다. ④번에서 갱신 대상이 되는 계좌 a1의 balance 값 역시 ②번과 ③번에 의해서 종료된 출금 트랜잭션이 갱신한 값이기 때문에 해당 고립화 수준을 만족시킨다. 그러나 ①번에서 읽은 balance 값과 ④번에서 갱신하려는 balance 값이 서로 달랐기 때문에 계좌 a1의 잔액이 음수가 되면서 일관성을 위반하였다. 결과적으로 ④번에서 비반복가능 읽기 현상이 발생하였기 때문에 이체 트랜잭션의 작업 결과가 데이터베이스의 일관성을 위반하게 된 것이다. 이와 같은 오류를 해결하기 위해서는 잘 못 설정된 고립화 수준을 수정해야 한다.

VI. 트랜잭션의 고립화 수준 시험 도구

1. 시험 도구의 구조

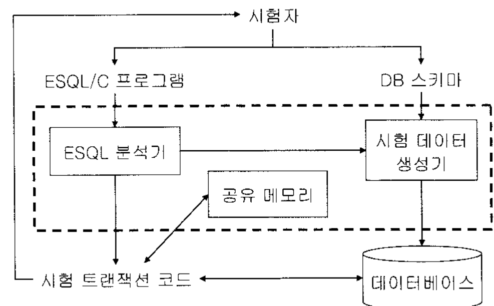

그림 4. 시험 도구의 구조

본 논문에서 제안하는 트랜잭션의 고립화 수준 시험 도구는 개발자가 많은 프로그램들을 작성하면서 고립

화 수준을 잘 못 설정하는 경우 발생하는 오류를 자동으로 검출하고자 한다. 제안하는 시험 도구는 SQL2를 지원하는 상업용 DBMS를 대상으로 ESQL/C로 작성된 프로그램을 시험한다. 시험대상이 되는 DBMS를 완전히 독립된 모듈로 취급하기 때문에 DBMS 자체 코드의 변경은 불필요하다. [그림 4]는 제안하는 시험 도구의 구조이다.

시험자는 개발자의 의뢰를 받아 ESQL/C로 작성된 프로그램을 시험하는 담장자이다. 점선 상자 내의 ESQL 분석기, 시험 데이터 생성기, 공유 메모리 등이 시험 도구를 구성한다. 시험자는 ESQL/C 프로그램을 ESQL 분석기에 입력하여 시험 작업을 시작한다. ESQL 분석기는 ESQL/C 프로그램의 SQL 문장들을 분석하여 고립화 수준을 검증할 수 있는 SQL로 구성된 문장들을 프로그램에 추가한다. 이와 같이 검증코드가 추가된 ESQL/C 프로그램을 시험 트랜잭션이라 하며 DBMS 환경 내에서 정상적인 트랜잭션과 동일하게 실행된다. 다음 절에서 시험 트랜잭션 코드를 생성하는 과정에 대해서 설명한다. 또한 ESQL 분석기는 시험에 필요한 데이터를 데이터베이스에 생성할 수 있도록 시험 데이터 생성기에 필요한 정보를 제공한다.

시험 데이터 생성기는 시험자로부터 트랜잭션 시험에 필요한 데이터베이스 스키마 정보를 받아서 데이터베이스에 필요한 테이블을 생성한다. 또한 ESQL 분석기로부터 시험 트랜잭션을 검증하는데 필요한 데이터에 대한 정보를 받아서 데이터베이스에 시험 데이터를 추가한다. 공유 메모리는 다수의 시험 트랜잭션이 동시에 실행되면서 동기화에 필요한 데이터와 시험 데이터에 대한 정보를 저장한다. 시험 트랜잭션의 실행이 종료되면 시험 결과를 시험자에게 제공하고 개발자가 오류를 수정할 수 있게 한다.

2. 시험 트랜잭션의 생성

2.1 검증 코드의 삽입

고립화 수준 설정 오류로 인하여 발생하는 문제인 손상가능 읽기, 비반복가능 읽기, 유령 튜플 등의 현상은 데이터베이스에서 데이터를 읽는 작업에서 발견될 수 있다. 따라서 SELECT, UPDATE, DELETE 등의 SQL 문장들을 실행할 때 세 가지 현상이 나타날 수 있다. 제안하는 시험 도구의 기본 개념은 시험 대상이 되는 ESQL/C 프로그램에서 이 SQL 문장들을 대상으로 세 가지 현상을 확인할 수 있는 코드를 삽입하여 고립화 수준 설정 오류를 검증한다. 다음 절부터 각 SQL 문장들이 고립화 수준을 검증하기 위한 코드가 삽입되는 과정을 기술한다.

2.2 SELECT 문장의 변환

ESQL/C 프로그램에서 SELECT 문장은 단일 튜플과 다중 튜플 검색에 사용되며 다중 튜플 검색의 경우 커서 방식을 사용한다. 본 절에서는 단일 튜플 검색은 커서 방식으로 쉽게 변환이 가능하므로 다중 튜플을 검색하는 경우만 다루기로 한다. 제안하는 시험 도구는 아직 프로토타입 수준으로 몇 가지 제약조건을 가진다. FROM 절에 단일 테이블을 가지는 SELECT 문장의 변환만을 고려한다. FROM 절에 다중 테이블을 가지는 SELECT 문장의 변환은 향후 연구 과제로 남기고자 한다. [그림 5]는 시험하고자 하는 프로그램의 다중 튜플 검색을 위한 커서 관련 문장들을 일반화하여 보여준다. 이후 지면 관계상 EXEC SQL을 [E]로 표기하기로 한다.

```
[E] DECLARE Csr CURSOR FOR
        SELECT a1, a2, ..., an FROM T
        WHERE condition;
...
[E] OPEN Csr;
while(NO_END_OF_DATA) {
    [E] FETCH Csr INTO :_a1, :_a2, ..., :_an;
    ...
}
```

그림 5. 시험할 SELECT 문장

테이블 T에서 조건식 condition을 만족하는 튜플들의 애트리뷰트 a1, a2, ..., an을 검색하는 트랜잭션이다. [그림 6]은 [그림 5]의 다중 튜플 SELECT 문장들을 검증코드로 대체한 시험 트랜잭션의 일부를 보여준다. 이후에 제시되는 [그림 6]과 같은 검증 코드들은 실제 삽입될 코드에 비해서 좀 더 추상화된 의사코드 형태로 표현한다. 밑줄로 시작하는 호스트 변수들에 대한 선언은 지면 관계상 생략한다.

2번 줄의 SELECT 문의 커서 정의에서 일부 애트리뷰트를 검색하는 대신 전체 애트리뷰트를 검색하도록 변형되었다. 6번 줄의 테이블 _T는 검색 대상이 되는 테이블 T의 이전 데이터를 저장한다. 5번 줄의 if 문장에 의해서 첫 번째 실행되는 SQL 문장일 때만 테이블 _T에 T의 모든 튜플들을 저장한다. 테이블 _T는 해당 시험 트랜잭션에서만 필요하기 때문에 다른 트랜잭션은 접근할 수 없다. 4번 및 9번 줄은 테이블 T와 _T의 데이터를 동일하게 유지하기 위해서 다른 트랜잭션이 테이블 T를 변경하지 못하게 한다. Protect_Modify_Begin(T)은 테이블 T에 대한 UPDATE, INSERT, DELETE 등의 SQL 문장을 실행하지 못하도록 운영체제의 IPC(inter process communication) 기능 중 세마포아를 사용하여 다른 트랜잭션들을 대기시킨다.

1	[E] DECLARE Csr CURSOR FOR
2	SELECT * FROM T
3	WHERE condition;
4	Protect_Modify_Begin(T);
5	if(firstSQL)
6	[E] INSERT INTO _T SELECT a1,...,an FROM T;
7	
8	[E] OPEN Csr;
9	Protect_Modify_End(T);
10	...
11	while(NO_END_OF_DATA) {
12	Protect_Commit_Begin;
13	[E] FETCH Csr INTO :_a1,:_a2,...,:_an,:_updPID;
14	Result = Tuple_Validation(_T, tuple, _updPID);
15	Protect_Commit_End;
16	Store(Result);
17	...
18	}

그림 6. SELECT 문장에 대해서 추가된 검증 코드들

14번 줄의 Tuple_Validation 함수는 테이블 _T와 현재 검색한 튜플 등을 통하여 고립화 수준을 검증한다. 14번 줄의 실인자명인 tuple은 13번 줄에서 검색한 애트리뷰트 값들을 하나의 튜플값으로 패킹한 레코드를 상징한다. 시험 트랜잭션들이 사용하는 각 테이블에 updPID라는 애트리뷰트가 추가된다. updPID는 각 튜플을 마지막으로 갱신한 트랜잭션의 프로세스 식별자(PID)이다. 12번 줄과 15번 줄의 Protect_Commit_Begin과 Protect_Commit_End는 데이터베이스에서 검색한 튜플을 Tuple_Validation 함수를 통해 검증하는 동안 테이블 T에 대한 변경을 한 트랜잭션들의 종료를 막기 위한 함수이다. 이 작업으로 트랜잭션 S가 변경한 튜플을 읽은 후 Tuple_Validation 함수를 호출하기 전에 트랜잭션 S가 종료함으로서 고립화 수준 검증을 불완전하게 할 가능성을 제거할 수 있다. Tuple_Validation의 호출 후 고립화의 수준 검증 결과가 반환되고 이를 Store 함수를 통해 결과 로그에 저장한다.

2.3 UPDATE 문장의 변환

```
[E] UPDATE T
    SET Exp(a)
    WHERE condition;
```

그림 7. 시험할 UPDATE 문장

UPDATE 문장은 기존에 저장되어 있는 튜플의 애트리뷰트 값을 수정하기 때문에 SELECT 문장과 유사하게 고립화 수준을 검증한다. [그림 7]은 시험하고자 하는 UPDATE 문장의 일반화된 형태를 보여준다.

[그림 7]의 UPDATE 문장은 테이블 T에 대해서 조건식인 condition을 만족하는 튜플들에 대해서 애트리뷰트 a의 값을 수식 Exp로 수정한다. [그림 8]은 이 UPDATE 문장에 검증 코드를 추가하여 변환한 문장들이다.

지면 관계상 [그림 6]에서 UPDATE 문장을 위해서 변경된 부분만 나타내기로 한다. UPDATE 문장을 위해서 [그림 6]의 15번과 16번 줄 사이에 [그림 8]의 문장들을 추가한다. [그림 8]에서 1번과 2번 줄에서 현재 검색한 튜플을 갱신한다. 현재 튜플을 시험 트랜잭션이 갱신하였기 때문에 현재 튜플의 updPID 값을 시험 트랜잭션의 프로세스 식별자인 PID로 저장한다. 또한, 3번과 4번 줄은 시험 트랜잭션의 다음 SQL 문장의 검증을 위해서 테이블 _T의 내용을 함께 반영해야 함을 보여준다. 테이블 _T에서도 테이블 T과 동일하게 현재 튜플을 갱신한다. keys(_T)는 테이블 _T의 주키를 표현하고 tuple.keys는 현재 튜플의 주키를 나타낸다.

1	[E] UPDATE T SET Exp(a), updPID = PID
2	WHERE CURRENT OF Csr;
3	[E] UPDATE _T SET Exp(a)
4	WHERE keys(_T) = tuple.keys;

그림 8. UPDATE 문장에 대해서 추가된 검증 코드들

2.4 DELETE 문장의 변환

DELETE 문장도 UPDATE 문장과 유사하게 기존에 저장되어 있는 튜플을 삭제하기 때문에 SELECT 문장과 유사하게 고립화 수준을 검증한다. [그림 9]는 시험하고자 하는 DELETE 문장의 일반화된 형태를 보여준다. [그림 9]의 DELETE 문장은 테이블 T에 대해서 조건식인 condition을 만족하는 튜플들을 삭제한다.

```
[E] DELETE FROM T
    WHERE condition;
```

그림 9. 시험할 DELETE 문장

[그림 10]은 이 DELETE 문장에 검증 코드를 추가하여 변환한 문장들이다. 지면 관계상 [그림 6]에서 DELETE 문장을 위해서 변경된 부분만 나타내기로 한다. DELETE 문장을 위해서 [그림 6]의 15번과 16번 줄 사이에 [그림 10]의 문장들을 추가한다.

```
1  [E] DELETE FROM T
2      WHERE CURRENT OF Csr;
3  [E] DELETE FROM _T
4      WHERE keys(_T) = tuple.keys;
```

그림 10. DELETE 문장에 대해서 추가된 검증 코드들

[그림 8]의 UPDATE 문장에 대한 검증 코드와 동일하게 현재 튜플을 테이블 T에서 삭제하고 테이블 _T에서도 해당 튜플을 삭제한다.

2.5 INSERT 문장의 변환

INSERT 문장은 새로운 튜플을 추가하기 때문에 사실상 고립화 수준을 검증하기 위한 코드를 추가할 필요가 없다. 그러나 시험 트랜잭션이 INSERT 문장을 포함하고 있다면 테이블 _T에 INSERT 문장이 삽입한 튜플을 동일하게 추가해야 다른 SQL 문장들의 검증을 완전하게 할 수 있다.

2.6 고립화 수준의 검증

검증 코드로 변환된 시험 트랜잭션이 고립화 수준을 검증하기 위해서 Tuple_Validation 함수를 호출한다. Tuple_Validation은 시험할 SQL 문장의 실행 이전의 테이블 T의 내용을 저장한 테이블 _T와 현재 검색한 튜플과 이 튜플을 최근에 갱신한 트랜잭션 식별자를 인자로 받아서 고립화 수준을 검증한다. [그림 11]은 의사코드 형태의 Tuple_Validation 함수를 기술한다.

```
1  int Tuple_Validation(_T, tuple, updPID) {
2    if(updPID ∈ Active_Trans_List) return(P0);
3    if(tuple_data(_T, tuple.keys) != tuple) return(P1);
4    if(tuple ∉ _T) return(P2);
5  }
```

그림 11. Tuple_Validation 함수

2번 줄의 Active_Trans_List는 공유 메모리에 유지되는 현재 실행중인 시험 트랜잭션의 프로세스 식별자들의 리스트이다. 3번 줄의 tuple_data 함수는 테이블 _T에서 tuple.keys의 주키를 가지는 튜플 데이터를 반환한다. 반환값인 P0, P1, P2 등은 각각 손상가능 읽기, 비반복가능 읽기, 유령튜플 등의 현상을 의미한다. 2번 줄에서 현재 튜플을 갱신한 트랜잭션이 아직 실행중이라면 손상가능 읽기에 해당한다. 3번 줄에서 현재 튜플의 주키값을 이용해서 테이블 _T에서 검색한 튜플과 동일한 데이터가 아니라면 반복가능 읽기를 위반한 것이기 때문에 비반복가능 읽기 현상에 해당한다. 4번 줄에서는 현재 튜플이 테이블 _T에 나타나지 않는다면 최근에 추가된 튜플이므로 유령튜플 현상에 해당한다.

3. 시험 도구의 구현

제안하는 고립화 수준의 검증을 위한 시험도구는 SUN Sparc Ultra-250의 Solaris 10 운영체제 환경에서 프로토타입 형태로 구현되었다. 구현에 사용된 범용 DBMS는 오라클 10.1이며 ESQL/C의 프로그래밍 환경인 ProC/C++로 구현하였다. 시험 트랜잭션들 사이의 통신 및 동기화 기능을 위해서 IPC(inter process communication) 기술의 공유 메모리(shared mem ory) 및 세마포아(semaphore)를 사용하였다. [그림 1]의 시험 도구를 구성하는 모듈 중 시험 데이터 생성기는 본 절의 프로토타입 구현에 포함하지 않았다. 시험 데이터 생성기는 고립화 수준의 검증에 중요한 역할을 하지 않기 때문에 향후 연구과제에서 구현하고자 한다.

고립화 수준의 검증 과정에 핵심 기능을 담당하는

ESQL 분석기는 ESQL/C 프로그램을 분석하여 각 SQL 문장을 검증 코드들로 대체한다. 분석 대상이 되는 SELECT, UPDATE, DELETE, INSERT 등의 문장에 대한 문법 구조를 파싱하여 검증 코드를 생성한다. 검증 과정에 필요한 호스트 변수들의 선언과 SQL 문장 실행전의 튜플들을 저장하는 테이블을 시험 트랜잭션 실행 초기에 생성하기 위한 코드도 추가한다.

공유 메모리에는 현재 실행되고 있는 시험 트랜잭션들에 대한 리스트인 Active_Trans_List가 저장된다. Active_Trans_List를 참조하여 튜플을 삽입했거나 수정한 트랜잭션이 종료되었는지를 결정한다. 또한, 세마포아는 Protect_Modify_Begin, Protect_Commit_Begin 등과 같은 다른 시험 트랜잭션과의 동기화를 위해서 사용한다.

4. 시험 도구의 평가

본 절에서는 프로토타입 형태로 구현된 고립화 수준 시험 도구를 실제 데이터베이스 응용 프로그램의 시험 과정에 적용하여 검증 결과를 평가한다. 구현한 시험 도구는 아직 프로토타입 형태이기 때문에 시험 데이터 생성기가 구현되지 않았다. 따라서 시험에 필요한 데이터베이스 스키마 및 튜플 데이터를 시험 트랜잭션 실행 전에 미리 데이터베이스에 저장하기로 한다.

평가 대상이 되는 데이터베이스 응용은 은행 데이터베이스와 관련된 은행 업무를 ESQL/C 프로그램들로 작성하였다. 시험 데이터베이스는 은행 업무에 관련된 5개의 테이블로 구성한다. 각 테이블은 주키와 필요한 경우 다른 테이블에 대한 외래키를 가진다. 5개 테이블은 평균적으로 100개 내외의 튜플을 저장하며 튜플 데이터는 각 테이블 내의 제약조건을 만족하도록 임의로 생성하였다. 시험할 ESQL/C 프로그램은 계좌 이체, 대출, 입금 및 출금, 계좌 생성 및 삭제, 직원 및 고객 정보 갱신 등 은행 업무에 관련된 작업을 한다. 총 생성된 ESQL/C 프로그램은 50개이며 고립화 수준 오류를 검증하기 위해서 고립화 수준 L0에서 L2까지로 임의로 설정된다.

[표 2]는 시험 도구의 프로토타입을 사용하여 50개 ESQL/C 프로그램을 시험한 결과를 나타낸다. 동시에 시험할 ESQL/C 프로그램의 수는 5, 10, 20, 30, 40, 50으로 할 때의 고립화 수준의 오류를 검출하는 비율이다. 동시에 실행되는 시험 트랜잭션이 5개일 때 39%의 오류 검출 비율을 보여줌을 알 수 있다. 이는 시험 트랜잭션 5개당 1.39개의 트랜잭션에서 고립화 수준의 오류를 검출할 수 있음을 의미한다. 시험 트랜잭션들 사이의 가능한 모든 스케줄을 실행하지 못 하기 때문에 100%의 오류 검출 비율을 달성하기는 어렵다. 그러나 동시에 실행되는 시험 트랜잭션의 수가 증가 할 수 록 오류 검출 비율이 상승하는 것을 볼 수 있다. 이는 시험 트랜잭션들 사이의 데이터 충돌이 자주 발생함에 따라서 고립화 수준의 설정 오류가 검출될 확률이 높아지기 때문이다.

표 2. 시험 도구의 오류 검출 비율

시험 트랜잭션 수	5	10	20	30	40	50
오류 검출 비율(%)	39%	42%	54%	65%	69%	73%

V. 결 론

과거 수십 년 동안 선언형 프로그래밍 언어로 작성된 프로그램에 대한 많은 시험 기법이 소프트웨어 공학 분야에서 연구되어 왔다. 그러나 많은 소프트웨어 시스템에서 핵심적인 역할을 하는 DBMS의 응용 프로그램들에 대한 시험 기법은 최근에야 활발한 연구가 진행되고 있다. 본 논문에서는 데이터베이스 응용 프로그램들의 고립화 수준 설정을 검증하기 위한 시험 기법과 프로토타입의 시험 도구를 구현하였다. ESQL/C로 작성된 응용 프로그램에 대해서 요구 조건에 부합하지 않은 고립화 수준으로 설정된 오류를 검출하기 위한 자동화된 시험 기법을 제안하였다. 또한 프로토타입의 형태로 구현된 시험 도구를 사용하여 은행 데이터베이스에 대한 성능 평가를 하였다. 50개의 은행 업무를 ESQL/C 프로그램으로 작성하여 예제 데이터베이스를 대상으로 고립화 수준의 오류 검출 비율을 평가하였다. 평가 결과는 동시에 실행되는 시험 트랜잭션의 수가 증가 할 수 록 고립화 수준 오류 검출 비율이 증가함을 보여주었다.

제안하는 시험 도구로 대량의 트랜잭션을 개발하는

시스템 환경에서 고립화 수준 설정 오류를 어느 정도 검출할 수 있는 방안을 제시하였다. 그러나 트랜잭션들 사이의 스케줄링 조합을 모두 실행할 수 없기 때문에 100%에 가까운 오류 검출 비율은 달성 할 수 없었다.

향후 연구과제로는 시험 도구의 한 모듈인 시험 데이터 생성기의 설계 및 구현을 할 예정이다. 또한 동시에 실행되는 시험 트랜잭션들 사이의 스케줄링 조합에 따른 오류 검출 비율을 향상시키기 위한 기법을 고안하고자 한다.

참고문헌

[1] B. W. Boehm, Software Engineering Economics, Englewood Cliffs, N. J.:Prentice Hall, 1981.

[2] D. Chays, Y. Deng, P. Frankl, S. Dan, F. Vokolos, and E. Weyuker, "An AGENDA for Testing Relational Database Applications," Software Testing, Verification and Reliability, Vol.14, No.1, pp.17-44, 2004.

[3] Y. Deng and D. Chays, "Testing Database Transactions with AGENDA," Proceedings of International Conference on Software Engineering, St. Louis, MI, pp.15-21, 2005.

[4] Y. Deng, P. Frankl, and Z. Chen, "Testing Database Transaction Concurrency," Proceedings of the 18th IEEE Internatioanl Conference on Automated Software Engineering, Los Alamitos, CA, pp.184-193, 2003.

[5] B. Daou, R. Haraty, and N. Mansour, "Regression Testing of Database Applications," Proceedings of the ACM Symposium on Applied Computing, Las Vegas, NV, pp.285-290, 2001.

[6] F. Haftmann, D. Kossmann, and A. Kreutz, "Efficient Regression Tests for Database Applications," Proceedings of Conference on Innovative Data Systems Research, pp.95-106, 2005.

[7] ANSI X3.135-1992, American National Standard for Information Systems - Database Language - SQL, 1992(11).

[8] J. Melton and A. R. Simon, Understanding The New SQL:A Complete Guide, Morgan Kaufmann, 1993.

[9] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Nei,l and P. O'Neil, "A Critique of ANSI SQL Isolation Levels," Proceedings of ACM SIGMOD, San Jose, CA, pp.1-10, 1995.

저자소개

홍 석 희(Seok-Hee Hong) 정회원

- 1989년 2월 : 홍익대학교 전자계산학과(공학사)
- 1991년 2월 : 한국과학기술원 전산학과(공학석사)
- 1997년 2월 : 한국과학기술원 전산학과(공학박사)
- 1997년 3월 ~ 8월 : 한국전자통신연구원(ETRI) 박사후 연수과정
- 1997년 9월 ~ 현재 : 경성대학교 컴퓨터정보학부 부교수

<관심분야> : 실시간 데이터베이스, 소프트웨어 테스팅, 트랜잭션 처리 시스템