관계형 데이타베이스 시스템에서의 사용자 정의 함수 지원

(User-defined Function Support in RDBMSs)

고 정 미 † 정 재 목 †† 김 형 주 †††

(Jeong-Mee Koh) (Jae-Mok Jeong) (Hyung-Joo Kim)

요 약 관계형 데이타베이스 시스템의 기능이 점차 확대되면서 사용자 정의 함수의 역할이 다양하게 되었다. 이에 따라 사용자 정의 함수의 수행 속도뿐 아니라 데이타베이스 시스템의 안정성과 보안의 중요성이 부각되었다. 사용자 정의 함수 지원방식은 크게 3가지로 나눌 수 있다: 정적 로딩 방식, 공유 라이브러리 방식, 프로세스 호출 방식. 기존의 데이타베이스 시스템에서 사용되는 공유 라이브러리 방식은 속도면에서 우수하나 안정성, 보안에 대한 요구를 만족시키지 못하며 시스템에 이식할 때 문제가 있다. 우리는 프로세스 호출 방식을 개선하여 사용자 정의 함수 지원을 구현하였다. 본 논문에서는 관계형 데이타베이스 시스템에서 사용하는 사용자 정의 함수를 구현하는 데 있어서 고려해야 할 점들을 살펴보고 성능을 측정한다. 성능 평가를 통해 우리가 구현한 방식이 공유 라이브러리방식과 성능차이는 적은 반면 장점이 많다는 것을 보인다.

Abstract As RDBMS became extended with new functionality, the role of user-defined functions(UDFs) has also been diversified. As well as its performance, DBMS's stability and database's security is more important than before. UDFs are supported using three approaches: static loading, shared library, and process invocation. Shared library approach used in traditional DBMS has an advantage of performance. But it has some flaws including stability, security, and portability. We implemented UDF support by improving process invocation approach. In this paper, we identify some interesting issues of supporting UDFs in RDBMS and measure the performance. Performance test indicates that the difference between our approach and shared library approach in performance is trivial, meanwhile our approach provides several additional advantages.

1. 서 론

사용자 정의 함수는 사용자가 데이타베이스에 함수를 추가함으로써 저장된 자료를 처리하기 편리하도록 시스템의 기능을 확장시키는 역할을 한다. 사용자 정의 함수는 사용하는 언어에 따라 SQL문으로 구현하는 질의 언어 함수와 C/C++같은 3GL 호스트 프로그래밍 언어를 이용한 프로그래밍 언어 함수가 있다. 보통 후자를 외부 함수(external function/routine)로 지칭하며 본 논문에서 언급하는 사용자 정의 함수(UDF: User Defined Function)는 이를 의미한다. 사용자 정의 함수는 3GL 언어의 강력한 프로그래밍 특성과 SQL의 자료를 쉽게 접근할 수 있는 장점을 통합할 수 있고 특정 응용이나 분야에 맞는 기능을 제공할 수 있는 확장성이 있다. 또한 코드의 중복을 피할 수 있고 여러 사람이 같이 공유할 수 있는 이점이 있다.

80년대 초반, 관계형 데이타베이스 시스템을 CAD등 엔지니어링 응용에 사용하기 위해 확장하려는 연구 결과, 이 새로운 응용에 맞는 새로운 타입 정의와 그에 대한 사용자 정의 연산에 대한 연구를 하게 되었다[1].

초기에는 정적 로딩 방법[2, 3]으로도 구현되었다. 이

† 학생회원 : 서울대학교 컴퓨터공학과
jmkoh@oopsla.snu ac.kr
†† 비 회 원 : 서울대학교 전산과학과
jmjeong@oopsla snu.ac kr
††† 종신회원 서울대학교 컴퓨터공학과
hjkim@oopsla.snu.ac kr
논문접수 : 1998년 11월 3일
심사완료 : 1999년 3월 27일

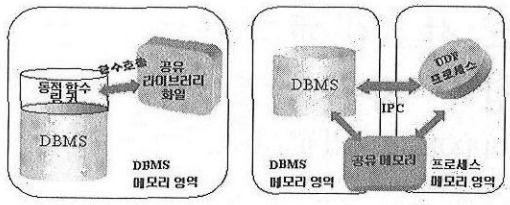


그림 1 공유라이브러리 방식과 프로세스 방식의 구조

방식은 사용자 정의 함수를 구현한 코드를 데이터 베이스 시스템에 링크하고 데이타베이스 시스템을 다시 부팅 해야 했다. 이 방법은 새로운 사용자 정의 함수가 추가될 때마다 데이타베이스를 재링크하고 부팅해야 하는 문제가 있어, 그 후에는 대부분 동적 로딩 방법을 사용하였다. 동적 로딩 방식은 실행 시 사용자 정의 함수의 등록이 가능하고 질의문에 사용된 함수에 대해 실행시간에 찾아서 수행하는 방식이다. 이 방식은 크게 공유 라이브러리 방식과 프로세스 방식으로 나뉠 수 있다. 공유 라이브러리 방식은 사용자 정의 함수를 공유 라이브러리에 등록하여 서버와 같은 프로세스 내에서 수행하는 방식이고[4,5,6,7] 프로세스 방식은 데이타베이스 시스템의 서버 프로세스와는 다른 프로세스를 두어 사용자 정의 함수를 수행하도록 하는 방식이다[그림 1]. 공유 라이브러리 방식은 성능이 뛰어나기 때문에 주로 사용된다.

관계형 데이타베이스를 확장하는 데 있어 사용자가 복잡한 내부 구조와 애트리뷰트를 가지는 새로운 자료형(data type)에 대해 정의할 수 있는 기능은 필수적으로 되었다. 그에 따라 사용자가 만든 복잡한 데이타나 멀티미디어 데이타, GIS 데이타 등에 대한 처리 또한 필요하게 되었다[8]. 다양한 용도의 사용자 정의 함수를 지원하는 데 있어 고려해야 할 점은 다음과 같다.

보안(security) 데이타베이스에 임의로 공개되지 말아야 하는 정보를 저장하는 일이 많아 보안은 데이타베이스에서 큰 이슈 중 하나다. 예를 들어 사용자가 의도적으로 데이타베이스 작업 공간에 대한 직접적인 메모리 접근을 하거나 권한에 대해 영향을 미치는 일이 발생하지 않도록 데이타베이스 시스템에서는 이를 막아야 한다. 또한 클라이언트/서버 형태의 데이타베이스 시스템에서 함수가 클라이언트 측에서 수행되는 경우 사용자의 개인 정보가 누출될 위험이 있다. WWW 환경에 데이타베이스 기술이 전개되면서 이러한 보안에 대한 요구가 최근에 더욱 높아지고 있다.

무결성(integrity)과 안정성(stability) 사용자 정의 함수의 코드가 점점 복잡해지면서 사용자 정의 함수 설계자의 실수로 인해 데이타베이스의 무결성이 파괴될 위험이 더욱 커졌다. 사용자 정의 함수의 광범위한 사용으로 고급 프로그래머일지라도 서버의 무결성이 손상되기 쉽다. 사용자의 실수가 있더라도 데이타베이스 시스템에 영향을 미치는 일은 없어야 하겠다.

독립성(independency) 데이타베이스 시스템이 운영되는 하드웨어나 운영체제에 의존적이지 않아야 한다. 데이타베이스 시스템의 용도가 다양해지면서 여러 플랫폼을 지원하는 시스템들이 많다. 독립성은 이러한 시스템에 있어서 중요한 문제이다. 또한 사용자가 사용자 정의 함수를 컴파일하는 환경도 다양하므로 컴파일러에도 의존적이지 않아야 한다. 하드웨어나 운영체제, 컴파일러에 의존적일수록 프로그래머가 해야 할 일이 많아지게 되므로 비효율적이다.

성능(performance) 위의 사항을 만족시키면서 많은 자료에 대한 사용자 정의 함수의 처리 속도도 빨라야 한다.

공유 라이브러리 방식의 경우, 데이타베이스 시스템의 주소 공간 내에서 함수가 수행되기 때문에 고의로든 실수로든 데이타베이스에 문제를 일으킬 수 있다. 그리고 다른 플랫폼을 지원하기 위해서는 그 플랫폼이 공유 라이브러리를 지원하는 형식에 맞도록 일일이 바꾸어야 하는 단점이 있다. 우리는 사용자 정의 함수 지원을 위해 서울대학교 객체지향 시스템 연구실에서 개발한 관계형 데이타베이스 시스템인 SRP[9]에 프로세스 방식을 이용하여 구현하였다. 이 방식의 장점은 위에서 나열한 네 가지 사항 중 처음 세 가지 사항을 포함한다. 그 외에도 심볼릭 디버거(symbolic debugger)의 사용이 가능하여 사용자 정의 함수 작성자에게 편리성을 주고[6] 다중 프로세서 방식의 시스템에서 수행되는 경우 함수 수행이 데이타베이스 시스템과 다른 프로세스에서 수행되므로 중앙 처리 장치의 이용률을 높일 수 있는 여러 장점을 지닌다. 그러나 이런 많은 장점에도 불구하고 주로 디버깅용으로 사용되었던[3] 이유는 성능상의 문제 때문이다. 프로세스의 성능 저하의 원인인 프로세스 생성과 프로세스간 통신시간을 줄이고 캐쉬를 이용하여 이 문제를 해결하도록 구현하였고 이에 대한 성능 측정을 통해 결론을 내릴 것이다.

논문은 다음과 같이 구성되어 있다. 2절에서는 사용자 정의 함수 지원을 위한 설계에 있어서 여러 가지 고려사항을 알아보고, 보다 효율적인 사용자 정의 함수의 지원을 설계하기 위한 방안을 제시한다. 3절에서는 사용자 정의 함수 지원을 위한 구현방식에 대해 살펴볼 것이다.

먼저 전체적인 구조와 개선 방안에 대해 이야기 할 것이다. 4절에서는 여러 사용자 정의 함수 지원방식에 대한 성능을 테스트하여 그 결과를 비교, 분석한다. 5절에서는 관련 연구에 대해 살펴보고, 마지막으로 6절에서 결론을 맺을 것이다.

2. 사용자 정의 함수의 지원 설계

2.1 사용자 정의 함수의 정의

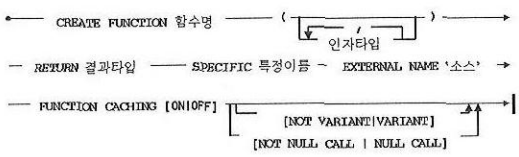


그림 2 CREATE FUNCTION 문

데이타베이스에 사용자 정의 함수를 새로 정의하기 위해 CREATE FUNCTION문[10]을 사용한다. SRP의 CREATE FUNCTION문은 그림 2와 같다. 여기에서 특정이름이 필요한 이유는 함수가 오버로딩 될 경우 때문이다. DROP FUNCTION 문에서 'DROP FUNCTION 함수명'으로 하면 없애고자 하는 함수뿐 아니라 오버로드된 함수가 모두 없어지므로 적절하지 않다. 각 함수에 대해 식별할 수 있는 특정이름을 주어서(이때 각 특정이름은 모든 함수에서 유일하다) 'DROP FUNCTION 특정이름'으로 해당하는 함수만 없어지도록 한다. 그림 2의 소스는 사용자 정의 함수의 소스파일이 있는 위치이다. FUNCTION CACHING은 사용자 정의 함수의 인자와 결과에 대해 캐싱여부에 대해 지정하는 것이다. 이에 관해서는 3절에서 자세히 설명된다. 예를 들어 C로 생성된 int udf(int arg)라는 사용자 정의 함수를 정의한다고 하자. 이 함수의 특정이름이 udf1이고 소스 파일명이 udf.c이며 /home/srp/udf/에 위치하고 이 함수에 대해 캐싱한다면 다음과 같이 사용자 정의 함수를 정의할 수 있다.

```
CREATE FUNCTION udf(integer) RETURN integer SPECIFIC udf1
EXTERNAL NAME '/home/srp/udf/udf.c' FUNCTION CACHING ON;
```

사용자 정의 함수의 결정성에 대해서는 SQL-92/PSM(Persistent Stored Modules)[11]에서 따라 사용자 정의 함수 구현자가 함수 등록 시 명시하도록 하였다. VARIANT/NOT VARIANT가 이에 대한 것이다.

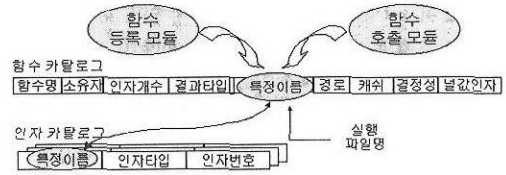


그림 3 함수 카탈로그와 인자 카탈로그

같은 인자에 대해 함수의 결과가 항상 같다면 NOT VARIANT이고 다를 수 있다면 VARIANT이다. VARIANT인 경우 캐쉬를 이용하지 않는다. NOT NULL CALL/NULL CALL은 함수 인자로 널(null)값이 들어왔을 때 사용자 정의 함수 수행 관리자가 어떻게 처리할 지에 관한 것이다. NOT NULL CALL로 선언하는 경우, 함수 인자가 널값이면 함수 수행을 하지 않고 널값을 반환한다. 그러므로 사용자가 사용자 정의 함수 코드를 쉽게 생성할 수 있으며 함수 호출을 피함으로써 성능이 향상된다. NULL CALL인 경우에는 사용자 정의 함수에서 널값의 함수 인자를 처리해야 한다.

2.2 고려 사항

관계형 데이타베이스에서 사용자 정의 함수 지원을 설계하는 데 있어서 고려해야 할 점들이 몇 가지 있다. 사용자가 사용자 정의 함수를 구현하고 등록하는 과정에서 인자의 개수나 자료형(data type)에 대해 제한을 받을 수 있다. 객체지향형 데이타베이스인 SOP[1]의 경우 이 문제에 대해 해결책으로 전처리 과정을 거치는 방법을 제시하였다[4]. 관계형 데이타베이스에서는 객체지향 데이타베이스와는 내부 함수의 특정이름뿐 아니라 해당 함수의 몇 번째 인자인지에 대해 인덱스를 구성하면 함수 카탈로그와 인자 카탈로그를 조인할 때 편리하다.

그리고 사용자의 편의를 위해 함수 오버로딩(overloading)에 대해 지원하는 것이 바람직하다. 즉 같은 함수명이라도 인자의 개수나 자료형이 다르면 이를 상이한 함수로 인식하고 처리해야 한다. 함수 등록할 때나 실행할 때 함수가 존재하는 지, 또 오버로드된 함수라면 그 중 어느 함수인지를 가려내는 과정(function resolution process)이 필요하다. 본 구현에서는 다음과 같은 과정을 거친다. 함수 카탈로그에서 함수명과 인자의 개수가 같은 함수를 먼저 찾는다. 이때 해당 함수가 하나 이상이면 첫 번째 인자부터 차례로 인자형에 대해 비교하여 해당 함수를 찾는다. 이때 함수 인자형에 대한 승격(datatype promotion)이 가능한지 살펴보아야 한다.

1) SNU OODBMS Platform :서울대학교 객체지향 시스템 연구실에서 개발하고 있는 객체지향형 데이타베이스 시스템

예를 들어 int형의 인자형을 갖는 함수가 정의되었고 실제 질의문에서 그 함수 인자로 smallint(C에서 short에 해당)형의 컬럼이 인자값으로 온 경우, 승격이 가능하므로 해당하는 함수가 존재하는 것이다. 이 경우 함수 호출과정에서 short를 int로 형변환(type casting)을 해주어야 한다. 반면에 컬럼의 자료형이 double이라면 승격이 불가능하므로 해당하는 함수는 존재하지 않는 것이다. SRP에서 자료형의 승격은 다음과 같다.

```
tinyint   -> smallint  -> integer  -> double
tinyint   -> smallint  -> integer  -> numeric
timestamp -> time, date
char      -> timestamp, time, date
```

이렇게 해당 함수를 찾으면 각 함수는 시스템 내부적으로 특정이름을 이용하여 오버로드된 함수들과 구분하므로 사용자가 이에 대해 알 필요가 없이 오버로드된 함수들을 사용할 수 있어서 편리하다.

2.3 실행 과정

사용자 정의 함수가 등록되는 과정은 다음과 같다. 사용자가 1절의 CREATE FUNCTION문을 입력하면 서버에서는 사용자가 만든 함수파일(*.c)을 실행파일로 생성하기 위해 함수를 호출하는 main() 모듈을 생성해야 한다. 이 작업을 하는 부분이 [그림 4]의 사용자 정의 함수 생성자이다. main() 모듈 생성 시 데이타베이스 시스템의 자료형을 C언어에 맞도록 사상(mapping)시키는 작업이 필요하고 main() 모듈 내에 프로세스간 통신을 하기 위한 코드가 필요하다. 예를 들어 데이타베이스 시스템의 타입시스템을 따라 smallint로 선언한 것을 해당하는 C 자료형, short로 사상(mapping)해야 한다. 예제 1의 age()함수를 정의한 age.c 파일이 있다면 함수 age()를 호출하는 main()모듈은 예제 1의 main()함수와 같이 생성된다.

그 다음 단계에서는 소스파일과 생성된 main()모듈을 하나의 파일로 생성한다. 이 파일을 컴파일하여 생성된 실행파일을 알맞은 디렉토리로 옮겨놓아야 한다. 사용자

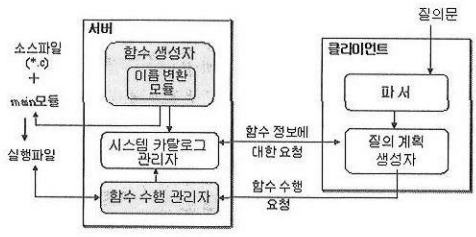


그림 4 사용자 정의 함수 호출

정의 함수 생성자는 이러한 일련의 과정을 하는 스크립트 파일을 만든다. 그래서 사용자가 이 스크립트 파일을 이용하여 오프라인으로 실행파일을 생성할 수 있도록 한다. 실행파일 생성 시 사용자 정의 함수 생성자에서 실행 파일명을 정하는 작업도 한다. 이 과정에서 복잡한 이름변환 루틴(mangling routine)을 거치지 않고 'main'과 특정이름을 합쳐서 유일한 이름을 갖는 실행파일을 생성한다. 등록된 사용자 정의 함수를 실행하는 과정은 다음과 같다. 사용자가 질의문을 통해 함수 수행을 요청하면 클라이언트의 파서가 파싱을 하면서 질의 계획 생성자가 질의문에 대한 수행 계획을 생성한다. 이 과정에서 함수에 대한 등록정보를 필요로 하기 때문에 서버의 시스템 카탈로그 관리자에게 이에 대해 요청한다. 클라이언트가 완성된 수행 계획을 서버에게 보내면 서버는 함수를 수행시킨다. 서버내에서도 함수(UDF) 수행 관리자[그림 4]가 등록할 때 생성된 실행파일을 수행함으로써 이 역할을 한다. 함수 수행 관리자에 대해서는 3절에서 더 자세히 설명할 것이다.

3. 사용자 정의 함수의 지원 구현

본 논문의 구현은 앞에서 말했듯이, 프로세스 방식을 사용한다. 사용자 정의 함수를 수행하는 과정에서 서버에서 수행하는 작업은 다음과 같다. 우선 서버는 수행할 사용자 정의 함수에 대해 공유 메모리를 할당하고 사용자 정의 함수를 수행하는 프로세스(이후 함수 프로세스)를 생성한다. 함수를 수행하는 데 필요한 함수 인자를 공유 메모리에 보내고 이를 알리기 위해 메시지 큐를 이용하여 메시지를 보낸다. 사용자 정의 함수 프로세스는 공유 메모리로부터 인자를 받아 함수 수행을 한 후 서버가 할당한 공유 메모리에 함수 결과값을 보낸다. 함수 프로세스가 작업을 마치고 시그널을 보내면 서버의 시그널 핸들러가 기다리고 있는 쓰레드를 깨우고 공유 메모리에서 결과를 가져오는 일련의 과정을 거치게 된다.

```
int age (int birth, year)
{
        char garbage[20];
        char current_year[5];
        time_t current = time(NULL);
        sscanf(ctime(&current), "%s %s %s %s %s", garbage,
garbage, garbage, garbage, current_year);
        return (atoi(current_year) - birth_year + 1);
}
int main (int arg, char* argv[])
```

```
{
  main함수의 인자로부터 메시지큐 식별자, 공유메모리 식별자를 구함
  공유 메모리 블록들중 이 함수의 인자 블록을 가져옴
  while ( Request.Type != CLOSE ) do
   메시지큐에서 함수 수행요청과 수행에 필요한 정보를 받음
   switch ( Request.Type ) do
     case ( 'EXECUTE' ) do          ▷함수 수행
      Ptr <- SharedMemoryArea;      ▷인자값에 대한 포인터
      While ( no_of_arg ) do
           int Arg;       ▷integer형 인자 컬럼에 대한 해당 C 언어타
                          입 int형으로 매핑
           Arg <- Ptr; ▷공유메모리의 인자값을 적절한 인자형으로
                          선언된 변수에 할당
           int Result <- age(Arg), ▷함수 호출
           ResultBuf <- Result;     ▷함수 수행 결과를 저장
           Ptr과 ResultBuf를 증가
      end while;
      서버에 메시지 보내고 수행이 끝났음을 알리는 시그널 보냄
     end case;
     case ( 'CLOSE' ) do    ▷프로세스 종료
         shutdown();
    end case;
   end switch;
  end while;
}
```

예제 1 사용자 정의 함수 생성 예

본 논문에서는 프로세스 방식의 단점인 성능상의 문제를 해결하고자 다음과 같은 방안을 제시한다.

3.1 블록단위의 이동

공유 라이브러리 방식과 프로세스 방식의 성능상의 차이는 크게 두 가지요인으로 살펴 볼 수 있다. 첫 번째 요인인 프로세스 생성시간에 대해서는 3절에서 해결안을 제시한다. 두 번째 요인은 서버 프로세스와 함수 프로세스간의 통신(IPC: interprocess communication)시간이다. 이 비용은 한번 통신할 때의 데이타 량과 통신 횟수에 따라 다른 데 한번 통신시의 데이타 량의 차이에 대한 시간은 큰 차이를 나타내지 않는다. 그 보다는 통신 횟수가 중요한 요인이다. SRP의 질의 수행 계획은 관계 대수(relational algebra)에 대한 표현으로 그 기본 단위인 SPROP(SRP Physical Relational OPerator)들이 노드(node)가 되는 방향성 그래프이다. SPROP은 물리적 연산자로 실제 질의 수행 루틴들의 집합을 가리킨다. 이 SPROP들이 대수에서의 연산자에 해당되며 테이블, 뷰 등이 피연산자이다. 그 종류에는 cartesian-product, hash-join, merge-join, filter, store, project, table-access, create-table, drop-table 등이 있다. 다음 예에 대한 SRP의 질의 수행 계획은 그림 5와 같다.

```
select   udf(relation1.col1)
from     relation1, relation2
where    relation1.col1 = relation2.col1;
```

모든 SPROP들은 공통 인터페이스로 Open(), Next(), Close()함수를 갖는다. Open()은 SPROP을 활성화시키는 함수이고 Close()는 SPROP을 비활성화시키는 함수이다. SPROP의 활성화를 위해 질의 계획 처리 모듈이 질의 수행 계획의 루트(root)인 최상위 SPROP의 Open()함수를 호출한다. 각 SPROP은 그 단계에 필요한 수행 준비를 하고 자신의 하위 SPROP의 Open()을 호출한다. 그러므로 최상위 SPROP의 Open() 함수 호출은 질의 수행 계획에 포함된 모든 SPROP들의 Open()함수의 호출로 이어져 결국 모든 SPROP들이 수행 준비를 마친다. 다음 단계에서는 demand-driven방식으로 Next()호출을 통해 실제 수행을 한다.

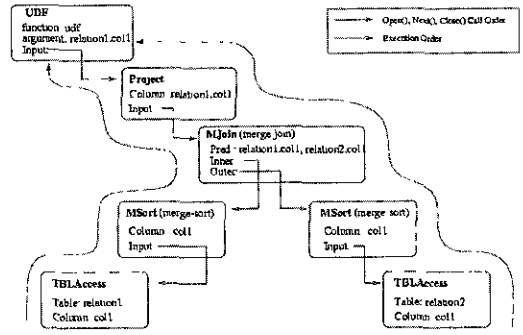


그림 5 질의 수행 계획의 예

커서 모듈은 한 튜플 씩 질의 계획을 수행하여 클라이언트에 전달할 버퍼에 그 결과를 채운다. 이 커서 모듈에서 질의 수행을 위해 최상위 SPROP의 Next()를 호출하여 수행을 요청한다. 각 SPROP은 자신에게 입력 값을 주는 하위 SPROP의 Next()를 통해 수행 요청을 하고 그 입력 값을 받으면 수행하여 자신의 상위 모듈에게 결과를 넘긴다. 그러므로 최상위 SPROP의 Next() 호출은 결국 리프(leaf)가 되는 최하위 레벨의 SPROP부터 수행을 하도록 한다. 각 SPROP은 한 튜플에 대해 수행하여 자신의 상위 SPROP으로 결과를 넘긴다. 커서 모듈에서는 최하위 레벨의 SPROP에서 더 이상 튜

플이 없을 때까지 Next()를 반복 호출한다.

서버 프로세스와 함수 프로세스간의 데이타 이동은 인자와 결과 값에 대해 이루어진다. 통신 횟수를 줄이기 위해 사용자 정의 함수 인자를 함수 프로세스에게 UDF SPROP 수행 시마다 보내지 않고 블록단위로 모아서 한번에 보내고 결과 값도 블록단위로 모아 한번에 받는 방식을 취하였다. 즉 UDF SPROP이 상위 모듈로부터 Next()를 통해 수행 요청을 받을 때마다 자신의 하위 SPROP에게 Next()를 호출하는 것이 아니고, 처음 Next()를 호출 받았을 때 하위 SPROP에게 더 이상 튜플이 없을 때까지 Next()를 반복 수행하여 그 결과를 가지고 있는다. 그 중 인자부분만 모아서 한번에 블록단위로 함수 프로세스에게 함수 수행을 요청하여 그 결과를 가지고 있는다. 이후 자신의 Next()가 호출되면 저장하고 있는 튜플을 하나씩 넘긴다. 예를 들어 UDF SPROP에서 Next()호출이 들어올 때마다 UDF SPROP을 수행하게 되는 경우 튜플의 개수가 10개이면 UDF SPROP은 함수 프로세스에게 수행요청을 10번 한다. 하지만 미리 Next()를 반복 수행하는 경우 한번에 10개의 인자를 보내므로 1번만 프로세스간 통신이 일어나 통신 횟수가 10분의 1로 줄어든다. 실제 SRP에서는 이 블록의 크기를 SRP 페이지 단위인 8K이므로 정수형의 단일 인자인 경우 한 블록에 최대 2048개의 인자가 들어가므로 한 블록만 이동한다고 해도 2048분의 1로 통신횟수를 줄일 수 있다. 4절의 성능 테스트에서는 블록당 인자의 개수를 조절하여 실험하였다.

3.2 함수 캐싱

이미지 관련함수, GIS 관련함수 등 복잡한 객체에 대한 처리요구가 증가함에 따라 함수 수행 시간이 길어졌다. 수행 시간이 긴 함수일수록 같은 함수에 대한 같은 인자인 경우 캐싱을 하여 함수 수행 횟수를 줄임으로써 얻는 성능향상은 크다. 간단한 함수일 경우에는 캐쉬 오버헤드 때문에 안 하는 것이 오히려 효과적일 수 있다. 이에 대해 사용자 정의 함수를 설계하는 사람이 CREATE FUNCTION문 선언 시 2절에서 설명했던 것처럼 캐쉬 여부를 결정할 수 있다. 캐쉬 크기에 대해서는 컨피규레이션에서 조절할 수 있도록 하였다. 캐쉬 기법에 대한 연구가 여러 가지 나와있는 데 본 구현에서는 간단히 LRU방식으로 구현하였다. 인자를 보내기 전에 사용자 정의 함수 캐쉬 관리자가 이미 캐쉬에 있는 인자인지 살펴보고 캐쉬에 있지 않은 인자에 대해서만 블록단위로 모아서 한번에 함수 프로세스에게 보낸다.

3.3 함수 프로세스 관리

앞에서 이야기하였듯이 프로세스 방식에서 성능상의

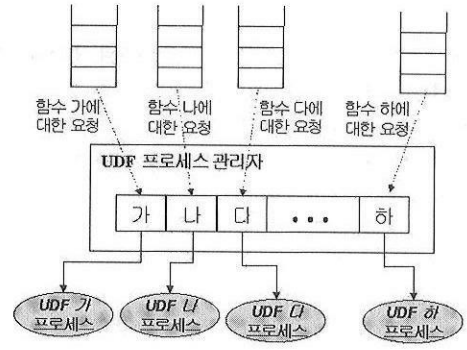


그림 6 프로세스 관리

차이를 가져오는 원인 중 하나가 프로세스의 생성시간이다. 공유 라이브러리의 경우는 한번 로딩하면 되지만 프로세스 방식에서 함수마다 프로세스를 생성하면 성능이 상대적으로 나쁠 수밖에 없다. 그래서 프로세스 생성시간을 절약하기 위해 사용자 정의 함수 프로세스 관리자는 함수 수행이 끝나고 함수 프로세스를 바로 없애지 않고 계속 유지시킨다. 사용자가 질의한 함수에 대해 이미 생성된 프로세스가 있으면 그 함수 프로세스에게 인자를 보내 함수를 수행하도록 한다. 즉 함수가 첫 번째 수행될 시에는 생성시간이 포함되지만 그 이후에 다시 같은 함수에 대해 수행 요청이 들어오면 이미 생성된 함수 프로세스가 수행하므로 프로세스 생성시간이 포함되지 않는다[그림 6]. 무한개의 함수 프로세스가 존재하는 것은 비효율적이므로 한 시점에 컨피규레이션에서 지정한 개수이하의 함수 프로세스가 같이 존재할 수 있도록 하였고 LRU방식으로 처리하였다.

개선된 구조에 대한 전체적인 그림은 그림 7과 같다.

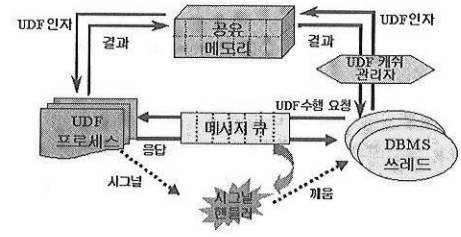


그림 7 개선된 방식의 UDF 수행 구조

4. 성능 평가

본 논문에서 제안한 사용자 정의 함수 지원 방안에

대한 성능을 알아보기 위해 서울대학교 컴퓨터 공학과 객체지향 시스템 연구실에서 개발한 관계형 데이타베이스 시스템 SRP(SNU Relational DBMS Platform)[9]에 구현하였다. 이 시스템은 C++로 구현된 클라이언트-서버형 데이타베이스 시스템이고 다중 쓰레드 방식의 단일 프로세스 서버로 구현되었다. 본 구현은 C++ 컴파일러 g++ 버전 2.8.1을 이용하였고 플랫폼은 Sparcstation 20(128M 메인 메모리), Solaris 2.5.1하에서 실험되었다.

4.1 성능 평가 모델

먼저 평가를 위해 사용된 여러 변수에 대해 설명한다.

4.1.1 사용자 정의 함수 모델

성능 측정을 위해 다음과 같은 질의문을 이용하였다.

```
select udf(arg1)
from   relation
where  <condition>;
```

프로세스 방식과 공유 라이브러리 방식, 두 방식의 함수 수행자체의 시간은 마찬가지이기 때문에 그 외의 비용을 줄여봄으로써 성능향상을 보이고자 한다. 그렇기 때문에 인자에 따라 수행시간이 변하는 것이 실제 상황에 가까울 수 있지만 인자에 의존하는 함수에 대해 실험하는 것은 별 의미가 없다. 단 함수의 수행시간은 영향을 미치기 때문에 이 udf()를 두 가지 함수로 구현하였다. 실행시간이 짧은 함수인 short_udf()와 비교적 긴 long_udf(), 각 경우에 대해 성능을 테스트하였다. 결과에 릴레이션(relation)을 읽어 들이는 시간도 포함되나 공유 라이브러리 방식이나 프로세스 방식이나 이 시간은 동일하다.

4.1.2 릴레이션

다음 릴레이션은 *short_udf()*에 대해 적용시킬 릴레이션으로 10,000개의 튜플로 구성되었다. 함수 캐쉬를 사용하는 경우를 고려하여 20%의 지역성을 가지도록 랜덤하게 발생시켰다.

- R10000 : 1 ~ 10,000까지의 값의 범위

다음 릴레이션 R200은 long_udf()에 대해 적용할 릴레이션으로 튜플 200개로 구성되었고 마찬가지로 함수 캐쉬를 사용하는 경우를 고려하여 20%의 지역성을 가지도록 랜덤하게 발생시켰다.

- R200 : 1 ~ 200까지의 값의 범위

4.1.3 기타 변수

- 함수 캐쉬 크기: R200을 적용하는 경우는 5, R10000의 경우는 50
- 블록수: SRP의 한 블록의 크기는 8K이나 함수 프로세스와 서버 프로세스간에 이동되는 블록 수를 늘려서 측정해보기 위해 한 블록 내에 들어가는 인자의 개수를 조절하였다. 다음은 데이타를 분석한 결과이다.

표 1 이동되는 데이타 블록수

	인자 블록수	결과 블록수
R200	5	1
R10000	12	5

4.2 성능 분석

4.2.1 지원 방식

이 실험을 통해 비교하는 대상은 표 2와 같다. 표에서 처음 4가지(pro1 ~ 4)는 프로세스 방식이고 나머지(lib1 ~ 2)는 공유 라이브러리 방식이다. 각 방식에 대해 실험하였다.

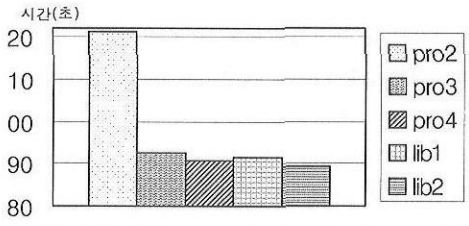
실험 결과에서 pro1방식일 때만 프로세스 생성시간이 포함되고 나머지 방식에는 프로세스 생성시간이 포함되지 않는다.

표 2 성능 평가 모델들

		프로세스 생성	프로세스간 통신	캐쉬
프로세스 방식	*pro1*	인자 당 생성	인자 당 발생	사용안함
	pro2	함수 당 생성	인자 당 발생	사용안함
	pro3	함수 당 생성	한번발생(블록단위)	사용안함
	pro4	함수 당 생성	한번발생(블록단위)	사용함
공유 라이브러리 방식	*lib1*			사용안함
	lib2			사용함

4.2.2 수행시간이 짧은 함수

R10000에 대해 pro2, pro3, pro4, lib1, lib2방식에 대해 *short_udf()*를 수행시켜보았다. 이에 대한 결과가 그림 8이다. pro1에 대해서는 10,000번의 프로세스 생성, 종료가 일어나므로 결과값 차이 때문에 R200에 대한 실험만 하였다. pro2에서는 사용자 정의 함수를 수행하는 SPROP(UDF SPROP)의 Next()함수가 호출될 때마다 하위 SPROP의 Next()함수를 호출하여 매 튜플에 대해 함수 인자값을 구해 함수 프로세스가 수행하도록 하는 방식으로 매 인자당 프로세스간 통신이 이루어진다. 반면에 pro3은 UDF SPROP의 Next()함수가 첫 번째 호출될 때 UDF SPROP에서 미리 Next()함수를 반복해서 호출해둠으로써 하위 SPROP에 대한 결과값을 저장하여 두고 그중 인자값 만을 한번에 함수 프로세스에게 보내 한번의 프로세스간 통신이 이루어지는 방식이다. 그림 8에서 이 두 가지 방식에 대한 결과가 첫 번째와



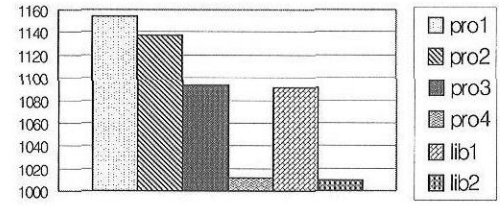
	pro2	pro3	pro4	lib1	lib2
시간(초)	121.09	92.67	90.61	91.41	89.39

그림 8 short_udf()에 대한 성능

두 번째 막대이다. 첫 번째 막대는 인자와 결과에 대해 각각 1개의 블록씩 10,000번의 프로세스간 통신이 일어난 결과이고 두 번째 막대는 인자에 대해 12개, 결과에 대해 5개의 블록에 대해 한번의 프로세스간 통신이 일어난 결과이다. 무려 28.42초의 차이가 났다. 프로세스간 통신의 횟수를 줄임으로써 성능향상이 크다는 것을 알 수 있다. 캐싱을 하지 않는 공유 라이브러리 방식인 lib1은 pro3보다 1.26초 가량 빨랐고, 프로세스 방식에 블록단위 이동과 함수 캐싱방법 둘 다 적용한 pro4는 블록단위 이동만 적용한 pro3보다 2.06초, 캐쉬를 하지 않는 공유 라이브러리 방식인 lib1보다 0.8초 가량 빨랐다. 프로세스 방식과 공유 라이브러리 방식 둘 다 함수 캐쉬를 적용한 경우에는 공유 라이브러리 방식이 1.22초 빨랐다.

4.2.3 수행시간이 긴 함수

수행시간이 긴 경우에 대한 성능 측정도 하였다. 캐쉬의 사용으로 수행 시간이 길수록 본 논문의 구현 방식의 이점이 크다는 것을 확인할 수 있다. 튜플이 200개인 R200에 대해 함수를 수행시켜 함수 호출횟수는 줄였다. 그림 9는 각 방식에 대해 수행시간이 약 5초 정도 되는 long_udf()를 수행한 결과이다. 매 Next()에 대해 함수



	pro2	pro2	pro3	pro4	lib1	lib2
시간(초)	1154.1	1136.9	1093.6	1011.4	1091.5	1009.6

그림 9 long_udf()에 대한 성능

프로세스 생성 및 종료, 프로세스간 통신이 이루어지는 pro1과 질의문의 한 함수 수행에 대해 단 한번의 함수 프로세스가 생성되고 마찬가지로 매 Next()시 프로세스간 통신이 이루어지는 pro2의 경우에 전자가 후자보다 17.2초 더 걸린다. 즉 200번의 함수 프로세스 생성과 종료시간이 17.2초이다. 200번의 프로세스간 통신이 일어나는 pro2와 1번의 프로세스간 통신이 일어나는 pro3은 43.3초의 차이가 난다. 199번의 프로세스간 통신비용이 이에 해당한다. 한 함수의 수행시간이 길수록 함수 캐싱의 이점이 커지므로 short_udf()에 대한 실험보다 pro3과 pro4의 성능 차이가 많이 난다. 또한 함수 캐싱을 하지 않은 공유 라이브러리 방식, lib1보다 함수 캐싱을 사용한 프로세스 방식, pro4가 80.1초나 빠르다. 함수 캐싱을 하는 공유 라이브러리 방식일 경우(lib2)는 pro4보다 1.8초 빠르다. 프로세스 방식, 공유 라이브러리 방식 둘 다 캐싱을 안 하는 pro3과 lib1는 2.1초, 둘 다 캐싱을 하는 pro4와 lib2는 1.8초 차이가 난다. 캐싱을 하는 프로세스 방식의 경우 캐싱을 하지 않는 프로세스 방식은 물론 캐싱하지 않는 공유 라이브러리 방식보다도 빠르고 둘 다 함수 캐싱을 하는 경우 더 성능 향상이 되지는 않지만 둘 다 안 하는 경우의 차이에 비해 성능 저하가 일어나지 않으므로 이를 구현함으로써 성능이 향상된다고 할 수 있다.

5. 관련 연구

현재 데이타베이스 시스템은 관계형 데이타베이스 시스템, 이를 확장한 객체-관계형 데이타베이스, 객체 지향 데이타베이스로 크게 나눌 수 있다. 객체 지향 데이타베이스 시스템인 Iris[6]의 경우는 데이타 모델이 함수형 데이타 모델이므로 애트리뷰트(attribute), 연관(association), 연산이 모두 함수로 구현된다. 그래서 다른 데이타 모델보다 훨씬 함수 사용빈도가 많기 때문에 약간의 성능상의 차이도 큰 차이를 빚어낼 수 있는 상황이어서 공유 라이브러리 방식으로 지원하고 있다. 관계형 데이타베이스에서는 객체 지향형 데이타베이스와는 모델이 다르기도 하고 그간 쌓아온 기술로 인해 보다 최적화기법이 발달하였기 때문에 별개로 고려해야 한다. 대표적인 객체-관계형 데이타베이스 시스템인 Oracle[12]에서는 다음과 같은 방식으로 사용자 정의 함수를 지원하고 있다. 서버 프로세스와는 다른 리스너 프로세스를 두어 서버가 사용자 정의 함수를 수행하는 경우 이 프로세스에게 요청을 한다. 사용자 정의 함수에 대해 질의하는 각 사용자마다 전용 함수 프로세스(extproc process)를 하나씩 두어 리스너 프로세스가 해당

되는 프로세스가 함수를 수행하도록 중개한다[13]. 본 논문에서는 3.3절에서와 같이 각 함수에 대해 하나의 프로세스를 두어 같은 함수에 대해 질의문을 수행할 경우 같은 프로세스가 수행을 하도록 하였다. DB2[10]에서는 사용자의 선택에 따라 데이타베이스 주소공간 내에서 수행되는 방식(not fencedmode)과 밖에서 수행되는 방식(fenced mode)을 지원한다. 하지만 fenced mode는 성능상의 문제로 인해 본 논문과는 달리 안정성을 위한 디버깅 용도위주로 사용된다. Illustra[5]와 UniSQL[14]에서는 공유 라이브러리 방식으로 지원하고 있고 인자의 개수나 타입에 대해 제한점이 있다. 이 외에 객체 지향 데이타베이스 시스템인 O2등에서 취하는 방식으로 클라이언트 프로세스에서 사용자 정의 함수를 수행하는 방식도 있다[15]. 이 방식은 불필요한 데이타의 이동량이 너무 많아 느리고 비효율적이어서[16] 논외로 한다.

표 3 데이타베이스 관리 시스템별 UDF 수행 방식

상품명	공유 라이브러리 방식	프로세스 방식	혼용 방식
Oracle			o
DB2	o	o	
Illustra	o		
UniSQL	o		

본 논문의 개선된 프로세스 방식은 성능면에서의 많은 향상으로 라이브러리 방식과의 성능상의 미비한 차이에 비해 보안성, 안정성, 독립성 등의 프로세스 방식이 가지는 여러 장점을 가질 수 있다. 따라서 사용자의 요구를 만족시키는 동시에, 확장성 증진 및 사용자 제약 제거 등의 이점도 지니게 된다.

데이타베이스의 보안이 중요해지면서 보안에 대한 다각도의 연구가 진행되고 있다 사용자 정의 함수에서도 마찬가지로 보안에 대한 여러 시도가 있다. 그 중의 한 갈래가 안전한 언어를 사용함으로써 보안문제를 해결하는 방법이다. 최근에 PREDATOR(PRedator Enhanced DAta Type Object Relational DBMS)[2][17] 객체-관계형 데이타베이스 시스템에서 이러한 접근 방식[18]을 취했다. 이 시스템에서는 사용자가 자바를 사용하여 사용자 정의 함수를 구현하도록 하고 데이타베이스 서버와 같은 프로세스 내에서 수행이 되는 방식을 취하였다. 자바는 언어특성상 자바 프로그램이 컴파일이 되기 전에 자바 바이트 코드라는 중간 코드로 생성된다. 이 코드는 플랫폼에 독립적이고 시스템에 해를 끼치지 않도록 하므로 데이타베이스의 무결성을 유지할 수 있다. 그러나 자바 코드가 느리기 때문에 이 방식에서도 성능이 문제가 된다. 자바가 각광받기는 하지만 아직은 C나 C++이 주요 언어로 사용되기 때문에 본 논문에서는 이들 언어에 적합한 해결책을 제시하였다.

2) PREDATOR시스템은 Cornell에서 개발한 객체 관계형 데이타베이스 시스템으로 Shore storage manager위에 질의 처리 엔진을 만들었다

6. 결론 및 향후 연구 계획

본 논문에서는 효율적인 사용자 정의 함수 지원을 위해 여러 방법들에 대해 살펴보고, 관계형 데이타베이스에서의 사용자 정의 함수를 지원할 시의 고려사항들을 제시하였다. 이를 토대로 하여 관계형 DBMS인 SRP에 개선된 프로세스방식을 이용하여 구현하였다. 본 논문에서 구현한 방식은 프로세스 방식을 이용하였기 때문에 이 방식이 갖는 장점을 다 가질 수 있다. 보안, 안정성, 독립성의 장점에 대해 1절에서 설명하였다. 그 뿐 아니라 독립성으로 인해 데이타베이스 시스템의 이식성이 향상되고 구현이 용이하다는 장점을 갖는다. 그러므로 기존의 관계형 데이타베이스 시스템에 사용자 정의 함수 지원 시스템을 추가적으로 구현하는 경우 큰 장점으로 작용하게 된다. 이 외에 시스템 카탈로그를 이용하여 인자의 개수나 형에 대한 제약과 함수 오버로딩을 쉽게 처리할 수 있다는 장점을 가지게 된다. 또한 4절에서 보였듯이 기존의 프로세스방식에 비해 큰 성능향상을 가져왔다는 측면에서 개선된 지원방식이라고 할 수 있다.

본 논문에서는 사용자 정의 함수의 수행자체에 대한 효율적인 지원방안에 대해 살펴보았다. 사용자 정의 함수의 성능상의 효율적인 지원방안의 다른 방향으로는 질의 최적화과정에 초점을 맞춘 연구가 많이 되어 왔다[19,20]. 향후 본 논문에서 구현한 작업에 적합한 최적화 기법에 대한 연구가 필요하다. 또한 사용자 정의 함수의 구현에 SQL문을 포함하는 것을 지원하는 것에 대한 연구가 필요하다. 이에 대해 지원할 경우, 데이타베이스 보안에 대한 주의가 더욱 필요할 것이다.

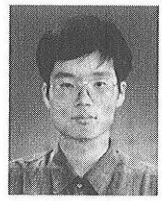
참 고 문 헌

[1] Michael Stonebraker and Brad Rubenstein and Antonin Guttman. "Application Of Abstract Data Types and Abstract Indices To CAD Databases," Data Week: Engineering Design Applications, pp 107-113, 1983. 5

[2] Guy M.Lohman and Bruce Lindsay and Hamid Pirahesh and K.bernhard Schiefer, "Extension to STARBURST: Objects, Types, Functions, and Rules," Communications of ACM, vol 34, no. 10,

1991. 10
[3] Michael Stonebraker, "Inclusion of New Types in Relational Data Base Systems," Proc. IEEE Int'l Conf. on Data Engineering, pp 262-269, 1986. 2
[4] 이병준, 김형주, "OODBMS를 위한 효율적인 메소드 지원 방안", Jounal of KISS(C) : Computing Practices, vol 4, no 5, 1998. 10
[5] Illustra Information Technology.Inc. Illustra User's Guide-Illustra Server Release 3.2, 1995. 10
[6] Kevin Wilkinson and Peter Lynbaek and Waqar Hasan, "The Iris Architecture and Implementation," IEEE Trans. on Knowledge and Database Eng., vol. 2, no. 1, 1990.3
[7] The POSTGRES95 User Manual, 1995.
[8] Judith R.Davis. Creating An Extensible, Object-Relational Data Management Environment: IBM's DB2 Universal Database, White Paper, DataBase Associates International, 1996. 11
[9] SNU OOPSLA Lab, SRP RDBMS PLATFORM 1.1 Manual, 1996
[10] Don Chamberlin, Using the new DB2, Morgan Kaufman Publishers, Inc., 1996.
[11] Andrew Eisenberg, "New Standard for Stored Procedures in SQL," ACM SIGMOD Record, vol 25, no 4, 1996. 2
[12] Oracle Corporation, PL/SQL User's Guide and Reference - Release 8.0, 1997, 6
[13] Oracle Corporation, Oracle8 Server Administrator's Guide - Release 8.0, 1998. 6
[14] The UniSQL User Manual, 1995.
[15] Sophie Garmerman, Personal Communication, O2 Technology, 1998.
[16] M.J.Franklin, Client Data Caching, Kluwer Academic Press, Boston, 1996.
[17] http://simon.cs.cornell.edu/info/Projects/PRREDATOR/docs.html
[18] Michael Godfrey and Tobias Mayr and Praveen Seshadri and Thorsten von Eicken, "Secure and Portable Database Extensibility," Proc. of the ACM SIGMOD Conf. on Management of Data, pp 390-401, 1998.
[19] Surajit Chaudhuri and Kyuseok Shim, "Optimization of Queries with User-defined Predicates," Journal of VLDB, pp 87-98, 1996.
[20] Joseph M.Hellerstein, "Predicate Migration: Optimizing Queries with Expensive Predicates", Proc. of the ACM SIGMOD Conf. on Management of Data, pp 267-276, 1993.5

고 정 미
1997년 2월 연세대학교 공과대학 컴퓨터과학과(학사). 1999년 2월 서울대학교 공과대학 컴퓨터공학과(석사). 관심분야는 객체-관계형 데이타베이스



정 재 목
1994년 2월 서울대학교 자연과학대학 계산통계학과(학사). 1996년 2월 서울대학교 자연과학대학 전산과학과(석사). 1996년 ~ 현재 서울대학교 자연과학대학 전산과학과 재학중. 관심분야는 트랜잭션처리, 멀티미디어 DBMS, 클라이언트-서버 DBMS

김 형 주
제 5 권 제 1 호(C) 참조