한국지리정보학회지 8권 1호 : 63~74(2005)

임베디드 시스템의 객체 관계형 DBMS에 적합한 공간 인덱스 방법 비교 연구

이민우[1※] · 박수홍[2]

Comparison research of the Spatial Indexing Methods for ORDBMS in Embedded Systems

Min-Woo LEE[1※] · Soo-Hong PARK[2]

요 약

차량 및 교통 분야의 대표적인 임베디드 시스템인 텔레매틱스 단말기는 대용량의 공간 데이터를 실시간으로 처리하기 위해서 RTOS(Real Time Operating System) 기반의 공간 DBMS를 요구하고 있다. 이러한 공간 DBMS는 기존의 ORDBMS의 사용자 정의 타입과 사용자 정의 함수라는 표준적인 기능을 이용하여 쉽게 확장 개발할 수 있지만, 공간 인덱스의 경우 SQL3에서 표준적인 개발 방법을 제공하지 않기 때문에, 임베디드 시스템과 같은 환경에서 공간 인덱스를 개발하는 것은 어려운 실정이다.

본 연구에서는 현재 ORDBMS에서 사용자 정의 인덱스를 개발할 수 있는 방법으로 제안되고 있는 Generalized Search Tree 방법과 Relational Indexing 방법을 비교·분석하고 각 방법에 대해 R-트리의 구현 및 영역 질의에 대한 실험을 통해 임베디드 시스템 환경에 적합한 공간 인덱스 방법을 제안하였다.

주요어 : 임베디드 시스템, 객체 관계형 DBMS, 공간 DBMS, 공간 인덱스, R-트리

ABSTRACT

The telematics device, which is a typical embedded system on the transportation or vehicle, requires the embedded spatial DBMS based on RTOS (Real Time Operating System) for processing the huge spatial data in real time. This spatial DBMS can be developed very easily by SQL3 functions of the ORDBMS such as UDT(user-defined type) and UDF(user-defined function). However, developing index suitable for the embedded spatial DBMS is very difficult. This is due to the fact that there is no built-in SQL3 functions to construct spatial indexes.

In this study, we compare and analyze both Generalized Search Tree and Relational Indexing meth

2005년 2월 2일 접수 Recieved on February 2, 2005 / 2005년 3월 28일 심사완료 Accepted on March 28, 2005
1 인하대학교 공과대학 지리정보공학과 Department of Geoinformatic Engineering, INHA University
2 인하대학교 공과대학 지리정보공학과 Department of Geoinformatic Engineering, INHA University
※ 연락저자 E-mail: lmana4@hotmail.com

ods which are suggested as common ways of developing User-Defined Indexes nowadays. Two implementations of R-Tree based on each method were done and region query performance test results were evaluated for suggesting a suitable indexing method of an embedded spatial DBMS, especially for telematics devices.

Keywords : Embedded System, ORDBMS, Spatial DBMS, Spatial Index, R-Tree

서 론

1.1 연구배경과 목적

무선 인터넷 및 이동 컴퓨팅 기술의 발달과 더불어 공간 데이터를 사용하는 응용 시스템들은 데스크 탑 환경뿐만 아니라 개인 정보 단말기, 텔레매틱스 단말기와 같은 임베디드 시스템 환경까지 그 활용 범위를 넓히고 있다. 임베디드 시스템이란 초소형 운영체제를 탑재하여 특정한 기능을 수행하도록 설계된 시스템으로써, 현재 이 분야의 핵심 기술은 초기의 임베디드 하드웨어 최적화 기술에서 임베디드 소프트웨어 최적화로 기술로 변하고 있다. 이에 따라, 차량 및 교통 분야의 대표적인 임베디드 시스템인 텔레매틱스도 하드웨어 단말기 보다는 대용량 공간 데이터의 실시간 처리와 효율적인 저장 및 관리가 가능한 Real Time Operating System(RTOS) 기반의 임베디드 공간 DBMS를 요구하고 있다. 데스크 탑 환경의 공간 DBMS가 기존 ORDBMS를 확장하여 쉽게 개발될 수 있듯이,(이상원, 2003) 임베디드 공간 DBMS도 기존의 임베디드 DBMS를 확장하여 개발이 가능하다. 하지만, 임베디드 DBMS 자체가 아직 개발 초기 단계이며, 최근 상용화되는 몇 안 되는 제품들은 개인 정보 관리 전용의 소형 RDBMS이기 때문에 이를 확장하여 공간 DBMS를 개발하는 것은 현재로는 어려운 실정이다. 따라서 임베디드 DBMS를 대신하여 데스크 탑 환경의 오픈 소스 DBMS를 분석하고 작게 수정 및 변경하여 임베디드 시스템에 적재하려는 시도가 이루어지고 있다.(Rajkumar Sen, 2004)

이와 같이 ORDBMS를 확장하여 공간 DBMS를 개발할 경우 공간 데이터의 저장 기법과 공간 연산자의 알고리듬 구현 이외에 반드시 고려해야 할 사항은 대용량의 공간 데이터로부터 특정 영역을 빠르게 검색하도록 도와주는 공간 인덱스의 구현이다(Guting, 1994). 현재 이러한 공간 인덱스는 데이터의 용량이 증가할수록 반드시 필요함에도 불구하고 대부분의 ORDBMS가 공간 인덱스와 같은 사용자 정의 인덱스(user-defined index)를 개발하는 방법을 제공하지 않기 때문에 구현 자체가 어려운 실정이다. 따라서 ORDBMS에서 사용자 정의 인덱스를 개발할 수 있는 일반적인 인덱스 개발 방법이 요구되고 있으며 현재 대표적으로 GiST(generalized search tree)(Hellerstein, 1995) 방법과 RI(relational indexing)(Hans-Peter, 2003) 방법이 제안되고 있다. 여기서 GiST는 ORDBMS에서 트리 기반의 사용자 정의 인덱스를 개발할 수 있게 해주는 Framework이며, RI는 ORDBMS에 이미 존재하는 인덱스 기법 및 SQL3의 확장성 기능을 이용하여 테이블 기반의 사용자 정의 인덱스를 개발하는 방법이다. 현재 이 두 방법은 ORDBMS를 확장하는 공간 DBMS에 일반적으로 적용 가능한 인덱스 개발 방법임에도 불구하고, 각각에 대한 비교·분석 및 성능 평가가 미흡하며 특히 임베디드 시스템 환경에서 평가된 사례는 없는 실정이다.

본 연구에서는 임베디드 시스템 기반에서 ORDBMS를 확장하여 공간 DBMS를 개발할 경우, 공간 인덱스를 개발할 수 있는 방법인 GiST와 RI를 비교·분석한 후 공간 인덱스 알고리듬을 적용하여 구현하고 영역 질의에 대한 실

험을 통해 임베디드 시스템 환경에 적합한 공간 인덱스 방법을 제시하는 것을 목적으로 한다.

1.2 연구내용과 방법

본 연구에서는 GiST와 RI를 비교·분석하여 공간 인덱스의 개발 방법들을 각각 연구하였으며, 동시에 이들 방법에 적용 가능한 R-트리 공간 인덱스 알고리듬(Gutman, 1984)을 연구하였다. 공간 인덱스의 개발 방법과 적용 가능한 알고리듬을 연구한 이후 Linux 운영체제 기반의 host와 QNX 운영체제 기반의 target으로 구성되는 임베디드 시스템 환경을 구축하였으며, 오픈 소스 DBMS인 PostgreSQL(The PostgreSQL Global Development Group, 2002)과 PostgreSQL의 공간 extension인 spatial processing engine(SPE)(Ho young Jung, 2004)를 target에 적재하여 공간 DBMS의 공간 인덱스 개발 환경을 구축하였다. 이러한 환경 하에 GiST와 RI를 이용하여 각각 R-트리를 구현하였다.

공간 인덱스의 영역 질의에 대한 실험 데이터로는 OGC(OpenGIS Consortium)의 “OpenGIS Simple Features Specification for SQL Revision 1.1”(Beddoe et al, 1999)을 준수하는 36000여개의 polygon 공간 객체를 사용하였다. 실험에 대한 성능 평가는 PostgreSQL의 interactive 클라이언트 프로그램인 psql에서 영역 질의를 요청하고 응답 받기까지 소요된 시간과 결과 레코드 수의 비교를 통해 공간 인덱스 개발 방법에 대한 성능의 타당성과 정확성을 평가하였다.

본 논문의 구성은 다음과 같다. 본론은 크게 2장과 3장으로 나누어지며, 2장에서는 사용자 정의 인덱스 개발 방법에 대해 살펴보고, 이에 대한 구체적인 구현 내용을 3장에서 논의하였다. 그리고 상기의 내용을 토대로 4장에서는 실험 및 분석을 기술하였으며, 마지막으로 이를 기반으로 결론을 도출하였다.

사용자 정의 인덱스 개발 방법

ORDBMS의 확장성을 대표하는 사용자 정의 타입과 사용자 정의 함수 생성 기능(Stonebraker, 1986)은 SQL3(A. Eisenberg, J. Melton, 1999)에 표준화 되어 있기 때문에, 모든 ORDBMS들은 이미지, 텍스트, 오디오, 공간 데이터와 같은 사용자 정의 타입을 DBMS 내부에 기본 데이터 타입처럼 저장할 수 있고 이들과 관련된 함수를 실행할 수 있게 된다. 즉, ORDBMS를 확장하여 이미지 DBMS나 공간 DBMS와 같은 응용 DBMS의 개발이 가능하게 된다. 하지만 응용 DBMS의 데이터 타입들은 기본 데이터 타입에 비해 그 크기가 방대하고 자료구조가 복잡하여 DBMS에서 수행되는 질의 처리 시간이 오래 걸릴 뿐만 아니라, 적절한 사용자 정의 인덱스를 사용하지 않고 일반적인 순차 탐색을 사용하여 질의를 수행하게 되면 효율적인 시간 안에 그 질의를 처리하지 못하게 된다. 따라서 ORDBMS를 확장하여 응용 DBMS를 개발할 경우 사용자 정의 타입에 적합한 인덱스 및 질의 처리의 최적화가 요구되고 있지만, 현재까지 이들을 개발하기 위한 SQL3의 표준화된 방법이 없는 실정이다. 따라서 상용 ORDBMS가 사용자 정의 인덱스를 개발할 수 있는 인터페이스를 제공하지 않으면 인덱스 개발이 원천적으로 불가능하게 되며, 오픈 소스 DBMS의 경우라도 DBMS 커널과 친숙한 개발자들을 제외하고는 DBMS 내부 구조를 파악하기가 쉽지 않다는 점과 불완전하게 사용자 정의 인덱스를 DBMS 커널과 결합하였을 경우 DBMS 자체에 악영향을 미칠 수 있다는 위험성 때문에 실제 사용자 정의 인덱스 개발은 어려운 실정이다(Hans-Peter, 2003). 이를 극복하고자 Oracle은 Data Catridge를, IBM DB2는 Data Extender를 제품으로 사용자 정의 인덱스 및 질의 처리 최적화 개발을 지원하지만, 이들이 제공하는 개발 방법은 특정 업체에 종속적이고 표준적이지 못하여 다른 ORDBMS와 상호

호환이 불가능하며 구현이 까다롭다는 단점이 있다.

결국 이미지나 공간 데이터와 같은 사용자 정의 타입을 DBMS의 기본 데이터 타입과 동등하게 사용하고 효과적으로 질의하기 위해서 사용자 정의 인덱스를 개발할 수 있는 보다 "일반적인" 개발 방법이 요구되고 있으며 현재 Generalized Search Tree와 Relational Indexing이 제안되고 있다.

2.1 Generalized Search Tree 방법

Generalized Search Tree는 B-트리, B+-트리, R-트리와 같은 DBMS의 다양한 탐색 트리들을 일반화시킨 search tree를 의미한다. GiST는 대부분의 DBMS 탐색 트리들이 B-트리 이후 거의 비슷한 형태로 확장되어 개발되는 점에 착안하여, 탐색 트리로써 가져야하는 Search key에 대한 자료구조와 노드 분할 메소드, 그리고 탐색 트리를 작동시키는 몇 개의 메소드들을 사용자가 정의할 수 있도록 노출시킴으로써 트리 기반의 사용자 정의 인덱스를 개발할 수 있게 하는 framework이다(Hellerstein et al, 1995).

보통 GiST는 2가지 형태로 ORDBMS에 적용되고 있는데, 첫 번째는 PostgreSQL과 같은 오픈 소스에 내부적으로 강하게 결합되는 형태이고, 두 번째는 GiST 라이브러리를 사용하여 상용 ORDBMS와 약하게 결합되는 형태이다. 두 번째 방법의 대표적인 예로는 Oracle의 Catridage와 GiST의 라이브러리를 결합한 OraGiST가 있다(Kleiner, 2003).

2.2 Relational Indexing 방법

RI는 DBMS 내부 구조를 확장하지 않고 SQL3의 Data Definition Language(DDL)나 Data Manipulation Language(DML)과 같은 DBMS의 기본적인 기능들만을 이용하여 사용자 정의 인덱스를 개발하는 방법을 의미한다. 즉, 일반적인 테이블을 마치 인덱스 테이블처럼 작동하도록 SQL 인터페이스를 확장하는 방법이다(Hans-Peter at al, 2003).

현재 RI는 DBMS의 내부 구조를 조작하지 않아도 된다는 장점 때문에 상용 ORDBMS에 많이 적용되어 사용되고 있다. 대표적으로 Oracle의 SQL 인터페이스를 확장하여 R-트리 및 X-트리, Linear Quadtree등을 구현한 사례들이 있다(Ravi Kanth at al, 1999).

2.3 Generalized Search Tree와 Relational Indexing의 비교 및 분석

GiST와 RI를 비교 및 분석하면, 표 1에서 알 수 있듯이 RI가 GiST보다 구현 측면과 적용 가능한 DBMS 측면, 그리고 적용 가능한 인덱스 기법 측면에서 보다 일반적이고 범용적인 것을 확인할 수 있다. 하지만 성능 측면에서 RI는 DBMS의 내부적인 운영을 그대로 따르기 때문에 테이블에 대한 트랜잭션과 locking의 부하를 기본적으로 가질 수밖에 없고 이에 대한 성능 저하가 필연적이다. 반면 GiST는 DBMS 내부 구조와 강하게 결합함으로써 이미 존재하는 인덱스들과 비슷한 성능을 가진다. 하지만 현재 이 두 방법의 성능을 실제적으로 비교 평가한 사례는 없으며 더욱이 임베디드 시스템 환경에서 성능을 비교한 연구는 없는 실정이다.

임베디드 시스템 환경의 공간 인덱스 구현

공간 인덱스 개발의 문제점을 해결할 수 있는 대안인 GiST와 RI를 살펴보고 분석하였다. 이번에는 본 연구에서 사용된 임베디드 시스템 환경과 공간 DBMS에 대해 살펴보고, GiST와 RI를 이용하여 R-트리를 구현하는 내용에 대해 기술하고자 한다.

3.1 임베디드 시스템 환경

TABLE 1. GiST와 RI의 비교 및 분석

구 분	대 상	내 용
구 현	GiST	· DBMS와 내부적으로 강하게 결합하는 부분이 매우 어려움 · 반면 노출된 메소드의 사용자 정의 코딩 부분은 쉬움
	RI	· DBMS 내부 구조를 고려할 필요가 없음 · 표준적인 SQL3 문법을 따르므로 구현이 쉬움
언 어	GiST	· 특정 프로그래밍 언어에 종속적임(현재 C/C++만 가능)
	RI	· 표준 SQL을 사용하므로 언어에 독립적임
성 능	GiST	· Built-in 인덱스보다는 떨어지지만 DBMS와 강하게 결합되므로 성능이 우수함
	RI	· 내부적인 연산 처리가 아닌 테이블의 relation 연산이 반복되므로 GiST보다 성능이 떨어짐 · 하지만 유연한 질의 및 DBMS 기능의 최적화로 보완
DB 수정	GiST	· DBMS의 내부 구조를 수정하므로 재 컴파일 필요
	RI	· DBMS 내부 구조를 전혀 수정할 필요가 없음
적 용 인덱스기법	GiST	· 트리 기반의 모든 인덱스 기법 적용 가능
	RI	· 트리뿐만 아니라 비 트리 기반의 모든 인덱스 기법 적용 가능
적 용 DBMS	GiST	· ORDBMS만 가능
	RI	· SQL3를 지원하는 모든 DBMS에 적용 가능
활 용	GiST	· DBMS와 강하게 결합하기 위해서는 오픈 소스 ORDBMS에만 적용 가능 · 상업용 ORDBMS와 결합하기 위해서는 GiST 라이브러리를 사용해야 하며 DBMS는 Oracle의 Catriadge와 같이 사용자 정의 인덱스 기능을 반드시 제공해야 적용이 가능함
	RI	·오픈 소스 및 상업용 DBMS에 모두 적용이 가능

본 연구에서 구축된 임베디드 시스템 환경은 표 2와 같다. host와 target의 프로세서는 모두 CISC 계열의 x86이며, 특히 target 프로세서는 텔레매틱스 단말기와 비슷한 사양의 PentiumⅡ 350MHz를 사용하였다. 일반적으로 이동 단말기에는 RISC 계열의 ARM7이나 XScale 등이 주로 사용되지만, RISC 계열의 프로세서는 CISC에 비해 메모리 사용이 비효율적이며 프로그램 측면에서 상대적으로 많은 명령어를 필요로 한다는 단점이 있다. 본 연구에서는 이동 단말기보다 상대적으로 높은 전력과 효율적인 메모리 관리를 필요로 하는 고사양의 텔레매틱스 단말기를 대상으로 하기 때문에, CISC 계열의 x86 프로세서와 64M 메모리를 사용하였다. 저장 매체로는 임베디드 소프트웨어로는 용량이 큰 DBMS를 포팅해야 하기 때문에 20G의 HDD를 사용하였다. 실제로 최근에는 많은 멀티미디어 응용 프로그램을 저장하기 위해 HDD를 내장한 텔레매틱스 단말기들이 많이 판매되고 있는 실정이다. 운영체제로는 실시간 처리가 보장되는 QNX Neutrino RTOS를 사용하였으며, 컴파일러로는 gcc 2.95를 사용하였다. QNX Neutrino

TABLE 2. 임베디드 시스템 환경

구 분	Host	Target
Processor	Intel Zeon 2.4 GHz	Intel PentiumⅡ 350MHz
Memory	1G	64M
HDD	40G	20G
OS	Linux(kernel 2.4.18-14)	QNX Neutrino
Compiler	gcc2.95 linux	gcc2.95.3 qnx-nto
ORDBMS	PostgreSQL 7.3.4	PostgreSQL 7.3.4
Spatial Extension	SPE 1.1	SPE 1.1

는 POSIX 표준을 따르며 실시간 마이크로커널 아키텍처를 기반으로 하는 임베디드 OS로써 국내·외 유수의 텔레매틱스 단말기 제조업체에서 사용되고 있다.

Target의 ORDBMS는 PostgreSQL 7.3.4를 QNX 운영체제에 적합하도록 포팅하였으며, 공간 DBMS는 PostgreSQL을 확장 개발한 SPE 1.1을 사용하였다. PostgreSQL은 오픈 소스 기반의 ORDBMS로써 SQL92와 SQL99를 지원하며, 동시성 제어나 트랜젝션 관리, 멀티 유저 기능과 같은 관계형 DBMS의 거의 모든 기능을 제공하고 있다. 특히, PostgreSQL은 사용자 정의 타입과 사용자 정의 함수와 같은 고수준의 확장성을 제공한다는 특징 때문에 개발 초기부터 지금까지 상업용 목적 보다는 연구 및 교육 목적으로 다양한 분야에 적용 및 실험되고 있으며, 본 연구에서 사용된 SPE도 PostgreSQL이 공간 DBMS 분야로 적용된 것이라 할 수 있다. 이러한 PostgreSQL을 Target의 임베디드 DBMS로 선정한 이유는 100% ANSI C 코드로 작성되었고 모두 공개되어 임의의 목적에 맞도록 코드를 수정 및 배포할 수 있으며 다양한 운영체제와 프로세서를 지원하기 때문이다. 실제로 PostgreSQL은 Linux, WinNT/Cygwin, Solaris, QNX등과 같은 20여개의 운영체제와 x86, Sparc, PPC750, arm32, MIPS등의 10여개의 프로세서를 지원하고 있다. 본 연구에 사용된 SPE는 OGC 표준을 준수하는 공간 DBMS로써 공간 데이터 저장 구조, 공간 연산자, 공간 인덱스, GML등과 같은 공간 DBMS의 전반적인 분야에 대한 연구 목적으로 개발되었다.

3.2 공간 인덱스 구현

본 연구는 이와 같은 환경에서 R-트리를 GiST와 RI 방법으로 SPE 공간 인덱스를 구현하였다. 다음 그림 1은 GiST와 RI로 개발된 인덱스의 구조를 나타낸다.

GiST를 이용하여 R-트리를 구현하기 위해 본 연구는 그림 2와 같은 OGC의 Geometry 타입(Beddoe et al, 1999)을 대상으로 하였고 Geometry의 MBR을 GiST의 key로 사용하였고, 트리를 구성하고 탐색하는데 필요한 6개의 메소

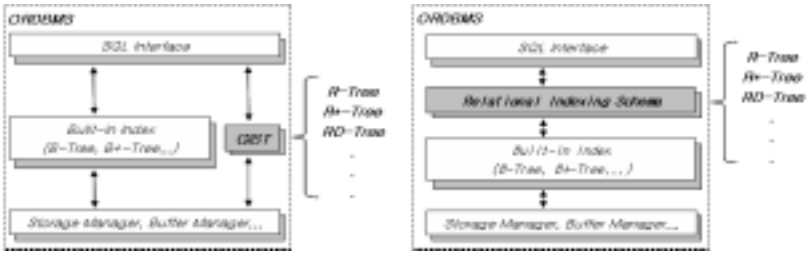

FIGURE 1. GiST와 RI의 인덱스 개발 구조

FIGURE 2. OGC의 Geometry 타입

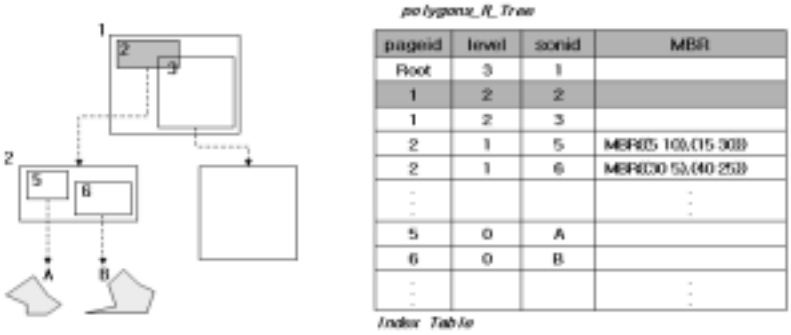

FIGURE 3. Polygon에 대한 R-Tree RI Index 테이블 구조

드들(consistent, union, compress, decompress, penalty, PickSplit)을 R-트리 알고리듬에 적합하도록 구현하였고, Node split 알고리듬은 Linear-cost를 사용하였다(Hellerstein et al, 1995).

그리고, RI를 이용해 R-트리를 구현하기 위해 RI의 3개의 기본 스키마 테이블(User 테이블, index 테이블, meta 테이블) 중 Index 테이블을 R-트리 구조로 구성하였다. RI 역시 OGC의 Geometry 타입을 대상으로 하였으며, Geometry의 MBR을 인덱스 되는 key로 사용하였다. RI는 테이블 간의 relational mapping 특성 때문에 반드시 노드의 최대 엔트리 수를 이용하여 Node Split 할 필요가 없다(Hans-Peter at al, 2003). 따라서 본 연구는 MBR 간의 중심점에 대한 거리를 이용하여 clustering quality를 측정하여 Split을 적용하였다(Kamel I at al, 1992). 다음 그림 3은 RI의 Geometry 타입 중 Polygon에 대한 R-트리 Index 테이블의 예를 나타내고 있다.

실험 및 분석

4.1 영역 질의 실험

본 연구는 데스크 탑 host의 PostgreSQL과 SPE를 임베디드 시스템 target에 포팅하고 GiST와 RI를 이용한 R-트리 인덱스 성능을 Target에서 실험하는 것이다. host에서 target으로의 포팅 과정은 host에서 최종적인 구현이 끝나면 target에 다운로드하는 교차 개발 방법을 이용하였다. 그 과정을 살펴보면, 먼저 host에 PostgreSQL을 target과 상호 호환 가능하도록 소스 컴파일 설치 후 그 위에 SPE를 더하였다. 그 후 host에서 SPE의 공간 인덱스를 GiST와 RI를 이용하여 R-트리를 구현 및 컴파일하고 실험에 사용될 polygon 데이터를 SPE의 spatial loader를 이용하여 DBMS에 로딩하여 host에서 target에 적재하기 적합한 물리적인 DB 파일과 중요한 공유 라이브러리를 생성한다. 마지막으로 이들을 target에 포팅하여 실험을 수행하였다.

실험 데이터로는 그림 4의 서울시 강남구 지가 현황도를 사용하였으며, 이는 OGC의 표준을 준수하는 36102개의 polygon 데이터로써 공간적인 집중도가 높기 때문에 영역 질의를 실험하기에 적합한 데이터이다. 그림 4의 Box는 총 20개의 영역 질의 MBR을 나타낸다.

질의들을 수행하고 처리하는데 측정된 시간은 PostgreSQL의 interactive 클라이언트 프로그램인 psql을 이용하여 영역 질의 명령을 서버로 보내고 서버가 처리해서 다시 그 결과를 psql에게 보내오는데 걸린 시간이며, 실험에서

는 20회 반복 측정된 시간의 평균을 사용하였다. 본 연구는 이와 같은 방법으로 target에서 순차 탐색, GiST, RI에 대해 각각 영역 질의를 수행하였고 실질적인 시간을 psql에서 측정하였으며, 각 질의의 수행 결과에 정확성을 측정하기 위해 질의 수행 결과의 count를 비교하였다.

FIGURE 4. 실험 데이터 및 임의의 영역 질의 (20개)

4.2 결과분석 및 고찰

표 3은 영역 질의를 순차 탐색으로 실험한 결과이다. 표 3에서 알 수 있듯이 순차 탐색으로 질의를 하게 되면 질의 처리 시간이 결과 레코드 수에 무관하게 거의 일정하게 소요됨을 확인할 수 있다. 그림 5는 이러한 표 3의 결과 레코드 수와 수행 시간간의 관계를 그래프로 나타낸 그림이다. 여기서 객체 수에 따른 약간의 질의 처리 시간의 차이는 단위 자체가 ms이기 때문에 거의 무시할 수 있는 수치이다.

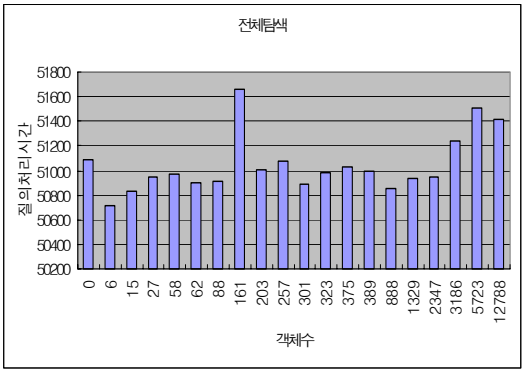

FIGURE 5. 순차 탐색의 레코드 수에 따른 질의 처리 시간(단위 ms)

표 4는 GiST를 이용한 R-트리 인덱스에 질의를 수행한 결과이다. 이를 분석해보면 우선 GiST에 대한 결과가 정확하게 검색된다는 것과 결과 레코드 수에 따라 질의 처리 시간이 약간 증가하는 것을 알 수 있다. 또한, 질의 처리 시간 자체가 전체적으로 매우 빠른 것을 확인할 수 있다. 특히, GiST의 공간 인덱스는 임베디드 시스템 환경에서 대략 2000여개 미만의 결과 레코드 수를 리턴하는데 1초도 걸리지 않는 좋은 성능을 보였다. 예를 들어 3번째 질의는 1329개의 결과 레코드를 리턴하는 영역 질의인데 순차 탐색의 경우 50931.21ms(약 50초)가 소요된 반면, GiST를 이용할 경우는 484.93ms(약 0.4초)

TABLE 3. 순차 탐색의 결과 레코드 수와 질의 처리 시간(단위 ms)

질의번호	레코드 수	수행 시간	질의번호	레코드 수	수행 시간
1	161	57658.18	11	323	50988.20
2	88	50909.21	12	375	51034.19
3	1329	50932.21	13	888	50857.22
4	257	51070.18	14	6	50716.24
5	27	50942.21	15	15	50835.22
6	301	50894.21	16	3186	51235.16
7	203	51003.19	17	2347	50945.21
8	389	50991.20	18	58	50968.20
9	0	51084.18	19	12788	51413.13
10	62	50896.21	20	5723	51506.12

TABLE 4. GiST의 결과 레코드 수와 질의 처리 시간(단위 ms)

질의번호	레코드 수	수행 시간	질의번호	레코드 수	수행 시간
1	161	384.94	11	323	344.95
2	88	143.98	12	375	313.95
3	1329	484.93	13	888	505.92
4	257	258.96	14	6	39.99
5	27	114.98	15	15	12.00
6	301	253.96	16	3186	1763.74
7	203	168.97	17	2347	1648.62
8	389	244.96	18	58	46.99
9	0	5.00	19	12788	5963.09
10	62	138.98	20	5723	1490.77

가 소요되었다. 그림 6은 표 4에 대한 레코드 수와 질의 처리 시간을 그래프로 나타낸 것이다. 그림 6의 점선으로 표시된 범례는 순차 탐색에 대한 결과이고 음영 처리된 범례는 GiST에 대한 결과이다. 전체적으로 순차 탐색보다 월등히 좋은 성능을 보였으며, 특히 작은 영역에 대한 질의에 대해서는 매우 빠른 속도를 보였다.

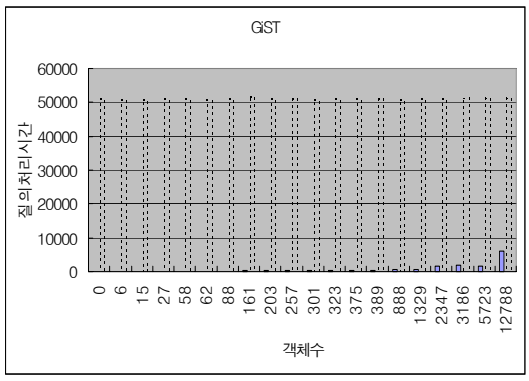

FIGURE 6. GiST의 레코드 수에 따른 질의 처리 시간

표 5는 RI을 이용한 R-트리의 영역 질의를 수행한 결과를 나타낸다. 이를 분석해보면, GiST와 동일하게 검색의 결과가 정확한 것으로 나타났으며, 결과 레코드 수에 따라서 질의 처리 시간이 증가하는 것을 확인할 수 있다. 전체적으로 GiST에 비해 성능이 떨어졌으며 결과 레코드 수가 매우 많아질 경우는 순차 탐색보다 도 느린 속도를 보였다. 또한, 결과 레코드 수에 10배 정도의 수치로 수행 시간이 계산되었음을 확인할 수 있었다. 그림 7은 표 5에 대한 레코드 수와 질의 처리 시간간의 그래프를 나타내고 있다. 그림과 같이 전체적으로 점선의 순차 탐색보다 빠른 속도를 보였으며, 결과 레코드 수에 따른 질의 처리 시간의 증가율이 매우 높음을 알 수 있었다. 예를 들어 100개 미만의 질의에 대해서는 1초 미만의 처리 시간이 소요되는 매우 빠른 속도를 보였지만, 1000개 레코드 수를 넘어서면서부터는 급격히 질의 처리 시간이 증가하는 것을 알 수 있다. 특히, 2000개를 넘게 되면 순차 탐색의 처리 속도의 50%를 넘었고, 6000여개에 대해서는 순차 탐색과 비슷한 성능을 보였으며, 그 이상에 대해서는 오히려 순차 탐색보다 느린 속도를 보였다.

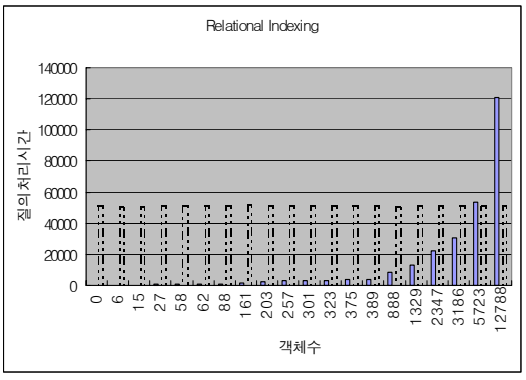

FIGURE 7. RI의 레코드 수에 따른 질의 처리 시간

TABLE 5. RI의 결과 레코드 수와 질의 처리 시간(단위 ms)

질의번호	레코드 수	수행 시간	질의번호	레코드 수	수행 시간
1	161	1773.73	11	323	3406.48
2	88	1032.84	12	375	3862.41
3	1329	12800.04	13	888	8617.68
4	257	2862.56	14	6	158.98
5	27	415.94	15	15	161.97
6	301	3034.53	16	3186	30377.35
7	203	2132.67	17	2347	22517.55
8	389	3857.41	18	58	801.88
9	0	668.90	19	12788	120897.50
10	62	138.98	20	5723	53473.82

그림 8은 500개 미만의 결과 레코드 수에 대해서 GiST(실선)와 RI(점선)의 질의 처리 시간을 나타낸 그래프이고 그림 9는 500개 이후의 결과 레코드 수에 대해서 질의 처리 시간을 나타낸 그래프이다.

그림 8과 그림 9에서 알 수 있듯이 전체 공간 데이터(36102개)에서 작은 영역을 질의할 경우(100개 미만)는 두 방법 모두 1초 이하의 만족할 만한 결과를 얻었다. 그리고 레코드 수가 100개 이상이고 800개 미만에 해당되는 적당한 영역 질의의 경우는 GiST(실선)는 여전히 1초 이하의 좋은 성능을 보였으며, RI(점선)은 10초 이하의 성능을 보였다. 마지막으로 800개를 넘어가는 영역 질의에 대해서는 GiST가 10초 이하의 성능을 보였고, RI는 결과 레코드 수의 10배 되는 시간의 성능을 보였다.

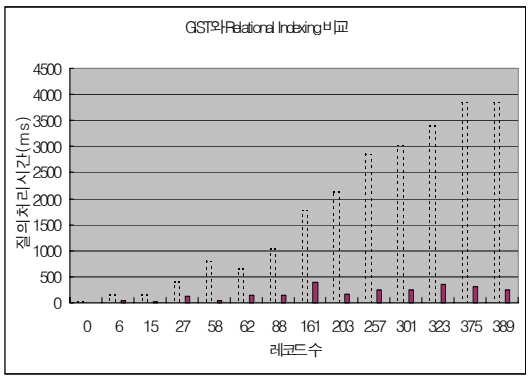

FIGURE 8. GiST와 RI의 비교(500개 미만)

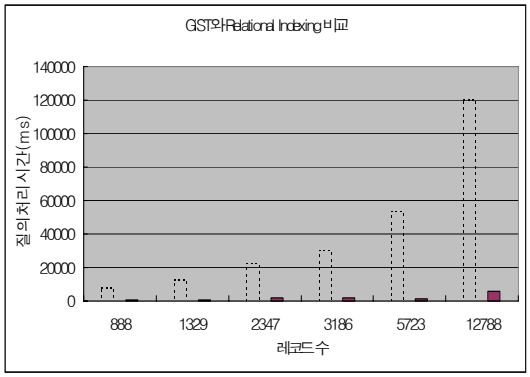

FIGURE 9. GiST와 RI의 비교(500개 이상)

결 론

본 연구는 임베디드 시스템에서 작동 가능한 ORDBMS를 확장하여 공간 DBMS를 개발할 경우 발생하는 공간 인덱스 개발의 문제점을 해결하고자 제안되고 있는 일반적인 사용자 정의 인덱스 방법인 Generalized Search Tree와 Relational Indexing을 살펴보았고, 실제 이 방법들을 이용해서 공간 인덱스를 구현하고 영역 질의에 대한 성능을 실험하였다.

연구의 실험 결과 Generalized Search Tree를 이용한 R-트리가 Relational Indexing을 이용한 R-트리 보다 영역 질의에 대해서 빠른 질의 처리 시간을 보이는 것으로 나타났다. 두 방법의 질의 처리 시간이 현저히 차이가 나는 가장 큰 이유로는 DBMS와 결합 구조의 차이인

것으로 사료된다. 즉, Generalized Search Tree는 PostgreSQL과 내부적으로 강하게 결합되어 있는 반면 Relational Indexing은 내부적인 결합이 전혀 없기 때문이다. 또 다른 이유로는 개발 언어의 차이를 고려할 수 있는데, 트리를 탐색하는 알고리듬의 순환적인 특성을 처리하는 능력이 Relational Indexing의 SQL 보다 Generalized Search Tree의 C 코드로 수행하는 것이 빠르기 때문이다. 하지만, 검색되는 데이터 양이 많지 않을 경우에는 Relational Indexing도 우수한 성능을 보였는데 이는 트리를 탐색하는 반복의 횟수가 적기 때문에 가능한 것으로 사료된다. 상기와 같은 분석 내용을 토대로 다음 결론을 얻을 수 있었다. 임베디드 시스템의 ORDBMS에서 Generalized Search Tree를 지원하고 트리 계열의 공간 인덱스를 개발해야 한다면 Relational Indexing 보다 Generalized Search Tree를 이용하는 것이 바람직하다. 하지만, 만일 임베디드 시스템의 ORDBMS가 Generalized Search Tree를 지원하지 않는다면 이 때는 클라이언트가 요구하는 질의 수준에 따라서 Relational Indexing의 적용을 고려할 필요가 있다. 클라이언트의 질의가 영역 질의가 아닌 Equal과 같이 전체 레코드 수에 대해서 극히 적은 레코드를 요청하는 질의라면 Relational Indexing을 사용하는 것도 좋은 방법이다. 또한, 반드시 트리 계열의 인덱스를 개발할 필요가 없다면 Space filling curve나 Spatial Hash와 같이 반복 연산이 적은 인덱스 기법을 Relational Indexing에 적용하는 것도 하나의 대안이라 할 수 있다.

참고문헌

이상원. 2003. 멀티미디어 데이터를 위한 확장형 인덱스 소개. 데이터베이스연구 19(3):1-9.

Eisenberg A and J. Melton. 1999. SQL:1999 formerly known as SQL3. ACM SIGMOD Record. 28(1):131-138.

Carsten Kleiner and U W. Lipeck. 2003. OraGiST-How to Make User-Defined Indexing Become Usable and Useful. Database Systems for BTW:324-334.

David Beddoe, P. Cotton, R. Uleman, S. Johnson and J. R. Herring. 1999. OpenGIS Simple Features Specification for SQL Revision 1.1. OGC.

Gutman, A. 1984. 6. R-Trees: A Dynamic Index Structure for Spatial Searching. Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston.:47-57.

Hans-Peter Kriegel, P. Marco and S. Thomas. 2003. The Paradigm of Relational Indexing : A Survey. Database Systems for BTW:285-304.

Jeung Ho young and Soo hong Park. 2004. A GML Data Storage Method for Spatial Databases. 인하대학교 대학원 석사학위논문:10-13쪽.

Hellerstein J.M, J.F. Naughton and A. Pfeffer. 1995. Generalized Search Trees for Database Systems. Proc. of the 21st Conf. on VLDB:562-573.

Kamel I. and C. Faloutsos. 1992. Parallel R-trees. Proc. of ACM SIGMOD Int. Conf. on Management of Data.:195-204.

Michael Stonebraker. 1986. Inclusion of New Types in Relational Database Systems. Proc. of ICDE:262-269.

Oracle Corp. 2000. Oracle9i Data Cartridge Developer's Guide Release 2. http://www.oracle.com/technology/documentation/oracle9i.html.

Rajkumar Sen. 2004. Efficient Data Management on Lightweight Computing Devices. Proc. of 4th ACM Int. Conf. on Embedded Software.

Ralf Hartmut Guting. 1994. An Introduction to Spatial Database System. VLDB Journal.

Vol. 3(4):357-399.

Ravi Kanth K. V, S. Ravada, J. Sharma and J. Banerjee. 1999. Indexing Medium-dimensionality Data in Oracle. Proc. of ACM SIGMOD Int. Conf. on Management of Data:521-522.

Stefan Debloch et al. 2001. Extensible Indexing Support in DB2 Universal Database.(In press) Components Database System. Morgan Kaufmann Publishers.

The PostgreSQL Global Development Group. 2002. The PostgreSQL 7.3.4 Administrator's Guide. http://www.postgresql.org.