creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

工學博士學位論文

위치 기반 서비스에서 이동 객체를 위한 연속 질의 색인 및 처리 기법

Indexing and Processing Continuous Queries over Moving Objects for Location Based Services

忠 北 大 學 校 大 學 院

情報通信工學科 情報通信工學專攻

朴 勇 勳

2011年 2月

工學博士學位論文

위치 기반 서비스에서 이동 객체를 위한 연속 질의 색인 및 처리 기법

Indexing and Processing Continuous Queries over Moving Objects for Location Based Services

指導敎授　柳 哉 秀

情報通信工學科　情報通信工學專攻

朴　勇　勳

이 論文을 工學博士學位 論文으로 提出함.

2011年　2月

Collection @ chungbuk

本 論文을 朴勇勳의 工學博士學位 論文으로 認定함.

審査委員長 조 기 형 ㊞

審査副委員長 안 재 형 ㊞

審 査 委 員 이 병 엽 ㊞

審 査 委 員 송 석 일 ㊞

審 査 委 員 유 재 수 ㊞

忠 北 大 學 校 大 學 院

2011年 2月

Collection @ chungbuk

차 례

차례 ····· i
표 차례 ····· v
그림 차례 ····· vi
국문 요약 ····· ix
영문 요약 ····· xi

Ⅰ. 서 론 ····· 1
1.1 연구 배경 ····· 1
1.2 연구 방법과 범위 ····· 7
1.3 본 논문의 구성 ····· 10

Ⅱ. 관련연구 ····· 11
2.1 디스크 기반 질의 색인 기법 ····· 13
2.1.1 Q-Index ····· 13
2.1.2 SINA (Scalable INcremental hash-based Algorithm) ····· 14
2.1.3 cGridex (Color Grid Index) ····· 20
2.2. 메모리 기반의 질의 색인 기법 ····· 24
2.2.1 CQI (Cell based Query Indexing) ····· 24

Collection @ chungbuk

2.2.2 VCR(Virtual Construct Rectangle) ····· 27
2.2.3 CES(Containment Encoded Squares) ····· 31
2.3 기존 메모리 기반 질의 색인 기법의 문제점 ····· 35
2.4 디스크 기반 질의 색인을 활용한 k-최근접 질의 처리 기법 ····· 37
2.4.1 SEA-CNN ····· 37
2.4.2 TB-kNN ····· 38
2.5 메모리 기반 질의 색인을 활용한 k-최근접 질의 처리 기법 ····· 39
2.5.1 YPK-CNN ····· 39
2.5.2 CPM ····· 41
2.5.3 CircularTrip ····· 43
2.6 기존에 제안된 연속 k-최근접 질의 처리 기법의 문제점 ····· 45

Ⅲ. 제안하는 연속 질의 색인 및 처리 기법 ····· 46
3.1. 연속 질의 처리를 위한 색인 기법 ····· 46
3.1.1 전체 색인 구조 ····· 46
3.1.2 질의 삽입 ····· 48
3.1.3 질의 삭제 ····· 51
3.2 연속 영역 질의 처리 기법 ····· 53
3.3 연속 k-최근접 질의 처리 기법 ····· 55
3.3.1 거리 관계 패턴 ····· 56
3.3.2 k-최근접 질의 처리 ····· 62

Collection @ chungbuk

3.3.3 객체 이동 시 연속 k-최근접 질의 처리 ····· 68
3.3.4 질의 이동 시 연속 k-최근접 질의 처리 ····· 74
3.3.5 다수의 객체 이동과 질의 이동에 따른 질의 처리기법 ····· 77

Ⅳ. 분석 ····· 80
4.1 PB-kNN의 저장 비용 ····· 80
4.2 CPM과 저장 비용 ····· 84
4.3 PB-kNN과 CPM의 저장 비용 비교 ····· 85
4.4 PB-kNN의 연산 비용 ····· 86
4.5 CPM의 연산 비용 ····· 89
4.6 PB-kNN과 CPM의 연산 비용 비교 ····· 90

Ⅴ. 성능 평가 ····· 91
5.1 연속 영역 질의 처리 기법 성능 비교 ····· 91
5.1.1 평가 환경 ····· 91
5.1.2 그룹 크기와 비트 식별자 크기에 따른 성능 변화 분석 ····· 92
5.1.3 그룹 크기에 따른 성능 비교 ····· 94
5.1.4 질의 크기에 따른 성능 비교 ····· 96
5.1.5 객체의 이동 속도에 따른 성능 비교 ····· 98
5.1.6 객체와 질의의 개수에 따른 성능 비교 ····· 99
5.1.7 성능 비교 분석 ····· 101

Collection @ chungbuk

5.2 k-최근접 질의 처리 기법 성능 비교 ····· 102
5.2.1 평가 환경 ····· 102
5.2.2 거리 관계 패턴에 따른 성능 변화 분석 ····· 104
5.2.3 그리드의 크기에 따른 성능 비교 ····· 105
5.2.4 질의와 객체 개수에 따른 성능 비교 ····· 108
5.2.5 k-값에 따른 질의 처리 시간 비교 ····· 110
5.2.6 객체 이동 속도에 따른 질의 처리 시간 비교 ····· 111
5.2.7 질의 이동 속도에 따른 질의 처리 시간 비교 ····· 112
5.2.8 객체 이동 량에 따른 질의 처리 시간 비교 ····· 113
5.2.9 질의 이동 량에 따른 질의 처리 시간 비교 ····· 114

Ⅵ. 결 론 ····· 115

참고문헌 ····· 116

감사의 글 ····· 116

Collection @ chungbuk

표 차 례

표 3.1 기호 설명 ····· 53
표 3.2 알고리즘 표현을 위한 기호 ····· 60
표 5.1 성능 평가 인자 ····· 89
표 5.2 성능 평가 인자 ····· 101

Collection @ chungbuk

그 림 차 례

그림 1.1 연속 질의 처리 기법의 구성 ····· 9
그림 2.1 Q-Index의 안전 영역 ····· 12
그림 2.2 SINA의 질의 처리 과정 ····· 15
그림 2.3 SINA의 해싱 단계 ····· 16
그림 2.4 cGridex의 칼라 ····· 19
그림 2.5 cGridex의 칼라 테이블 ····· 20
그림 2.6 셀 기반 질의 색인의 구조 ····· 24
그림 2.7 VCR의 구성 ····· 26
그림 2.8 질의를 분할하는 다양한 크기의 VCR ····· 27
그림 2.9 VCR의 검색 ····· 28
그림 2.10 CES의 구성 ····· 30
그림 2.11 CES들이 나타내는 트리 구조 ····· 30
그림 2.12 CES들의 계층관계 ····· 31
그림 2.13 CES에 의한 질의 분할 ····· 32
그림 2.14 객체의 이동에 의해 영향을 받은 질의 연산 ····· 34
그림 2.15 YPK-CNN을 이용한 2-최근접 질의 처리 ····· 38
그림 2.16 CPM을 이용한 2-최근접 질의 처리 ····· 40
그림 2.17 CircularTrip을 이용한 2-최근접 질의 처리 ····· 42
그림 3.1 제안하는 색인의 전체 구조 ····· 45

Collection @ chungbuk

그림 3.2 질의 삽입 알고리즘 (*InsertQuery*) ····· 47
그림 3.3 *PL*에 질의 삽입 알고리즘 (*InsertQueryIntoPL*) ····· 48
그림 3.4 *PL*에 질의 삽입 ····· 48
그림 3.5 질의 삭제 알고리즘 (*DeleteQuery*) ····· 50
그림 3.6 *PL*에서 질의 삭제 알고리즘 (*DeleteQueryFromPL*) ····· 50
그림 3.7 *PL*에서의 질의 삭제 ····· 50
그림 3.8 연속 영역 질의 처리 알고리즘 (*CQProcessing*) ····· 53
그림 3.9 *CLA*와 *CLR*의 차이 ····· 55
그림 3.10 N_{clr}에 따른 거리 관계 패턴 ····· 56
그림 3.11 최대 오차 E 계산 ····· 58
그림 3.12 거리 관계 패턴을 이용한 3-최근접 질의 처리 ····· 62
그림 3.13 *search-dist*를 고려한 검색 영역 ····· 63
그림 3.14 k-최근접 질의 처리 알고리즘 ····· 65
그림 3.15 연속 k-최근접 질의 처리를 위한 자료구조 ····· 65
그림 3.16 질의 q의 *k-dist*안으로 객체 o가 이동한 경우의 알고리즘 ····· 67
그림 3.17 질의 q의 영역으로 객체 o_4의 이동 ····· 68
그림 3.18 질의 q의 *k-dist* 밖으로 객체 o가 이동한 경우의 알고리즘 ····· 70
그림 3.19 질의 q의 영역 밖으로 객체 o_3의 이동 ····· 71
그림 3.20 식 3.8을 만족하는 질의 이동 예제 ····· 72
그림 3.21 질의 q가 한 VDC영역 내에서 이동한 경우의 알고리즘 ····· 74
그림 3.22 질의 q가 한 VDC영역 밖으로 이동한 경우의 알고리즘 ····· 74
그림 3.23 다수 객체가 이동한 경우의 알고리즘 ····· 75

Collection @ chungbuk

그림 3.24 다수 객체 이동과 질의 이동을 고려한 알고리즘 ····· 77
그림 5.1 비트 크기별 처리 속도 ····· 91
그림 5.2 비트 크기별 저장 공간 ····· 91
그림 5.3 그룹 크기별 저장 공간 ····· 93
그림 5.4 그룹 크기별 처리 속도 ····· 93
그림 5.5 최대 질의 크기별 저장 공간 ····· 95
그림 5.6 질의 크기별 처리 속도 ····· 95
그림 5.7 객체 이동 속도별 처리 속도 ····· 96
그림 5.8 질의 개수별 처리 속도 ····· 98
그림 5.9 객체 개수별 처리 속도 ····· 98
그림 5.10 N_{clr}에 따른 저장 비용과 질의 처리 비용 ····· 102
그림 5.11 그리드 크기에 따른 질의 처리 속도의 변화 ····· 105
그림 5.12 그리드 크기에 따른 저장 비용 ····· 105
그림 5.13 객체 개수에 따른 질의 처리 시간 변화 ····· 106
그림 5.14 질의 개수에 따른 질의 처리 시간 변화 ····· 107
그림 5.15 k 값에 따른 질의 처리 시간 변화 ····· 108
그림 5.16 객체 이동 속도에 따른 연속 질의 처리 시간 변화 ····· 109
그림 5.17 질의 이동 속도에 따른 질의 처리 속도 ····· 110
그림 5.18 객체 이동 량에 따른 질의 처리 시간 변화 ····· 111
그림 5.19 질의 이동 량에 따른 질의 처리 시간 변화 ····· 112

Collection @ chungbuk

위치 기반 서비스에서
이동 객체를 위한 연속 질의 색인 및 처리 기법

朴 勇 勳

忠北大學校 大學院 情報通信工學科 情報通信工學專攻

指導敎授 柳 哉 秀

요 약

최근 휴대용 모바일 기기의 발전과 위치 인식 기술의 발전으로 인하여 위치 기반 서비스에 대한 관심이 급격하게 증가하고 있다. 위치 기반 서비스는 주로 이동 객체를 대상으로 하고, 이러한 객체의 이동은 질의 결과에 영향을 미친다. 빈번한 객체 이동에 따른 효율적인 질의 처리 기법은 위치 기반 서비스에서 중요한 연구 분야로서 관심이 집중되고 있다. 실시간 질의응답 서비스를 위해 메인 메모리 기반의 효율적인 연속 질의 처리를 위한 질의 색인 기법과 연속 질의 처리 기법이 요구된다. 본 논문에서는 메인 메모리 환경에 적합한 그리그 구조 기반의 연속 질의 처리를 위한 색인 기법을 제안한다. 연속 영역 질의 처리를 위해 기존 기법은 질의 영역과 겹치는 모든 셀에 객체 식별자를 기록하기 때문에 중복적인 질의 식별자가 기록되는데 따른 저장 공간의 낭비

Collection @ chungbuk

가 발생한다. 또한 연속 질의 처리를 위해 객체 이동시 영향을 받는 질의를 판단하기 위해 질의 이전 위치와 현재 위치에 대한 영향을 미치는 질의를 비교할 때 질의 식별자를 이용하여 비교하기 때문에 연산비용이 크다. 제안하는 기법은 비트 패턴을 이용하여 질의 식별자 기록에 따른 저장 공간의 낭비를 방지하였고 또한 비트 패턴 비교를 통해 객체 이동이 영향을 미치는 질의를 찾기 위해 소모되는 연산 비용을 절약하였다. 연속 k-최근접 질의를 빠르게 처리하기 위해서는 질의 점으로 부터 가장 근접한 셀들을 순서대로 탐색해야 한다. 가장 근접한 셀을 효율적으로 연산하기 위해 셀 간의 거리 관계를 나타내는 거리 관계 패턴을 제안한다. 거리 관계 패턴은 모든 질의가 공유하기 때문에 연속 k-최근접 질의 처리를 위해 각 질의가 보조적인 데이터를 유지하는데 소모되는 비용을 절약하고 이미 순서화된 패턴을 활용하기 때문에 주변 셀들의 근접한 순서를 결정하는데 드는 연산비용을 절약한다. 본 논문에서는 다양한 성능평가를 통해 연속 질의 처리를 위한 색인 기법 및 질의 처리 기법의 우수성을 입증한다.

Collection @ chungbuk

Indexing and Processing Continuous Queries over Moving Objects for Location Based Services

Park, Yong Hun

Department of Information & Communication Engineering

Graduate School, Chungbuk National University

Cheongju, Korea

Supervised by Professor Yoo, Jae Soo

Abstract

Recently, the interests on Location Based Services(LBS) have been highly increased due to the development of the mobile devices and the positioning technologies. Continuous query processing is one of the important research issues since LBS is based on moving object environments. Continuous query processing schemes are required to provide real-time services. In this dissertation, we propose an efficient continuous query processing scheme using a modified grid based query indexing to reduce storage spaces and to accelerate the query processing time. The proposed scheme has two major features. First, each query has a bit

Collection @ chungbuk

identifier and each cell in a grid has a bit pattern that consists of the bit identifiers of the queries. The bit patterns represent the relationship between cells and queries. Using the bit patterns, we compute quickly what query overlaps a cell in a grid and reduce the number of unnecessary operations by comparing the bit patterns instead of the query identifiers when we compute the relation between cells and a query. Second, the management of cells in the grid by groups prevents from wasting the storage space through the increase of the length of the bit pattern and increasing the comparison.

A continuous k-NN(k-Nearest Neighbor) query is one of the most important query types for LBS. The existing methods for processing the continuous k-NN query have problems that they access additional cells unnecessarily and spend the high computation cost to find the nearest cells on a grid index. In addition, they do not consider the movement of a query. In this paper, to overcome the problems, we propose a new scheme to process the continuous k-NN query using the patterns of the distance relationship among the cells in a grid index. We normalize the distance relationship as certain patterns. The proposed scheme saves computation costs relatively over the existing schemes. It is shown through the various experiments that our proposed scheme is much better than the existing schemes in terms of the query processing time and the storage overhead.

---

* A thesis for the degree of Doctor in February 2011.

Collection @ chungbuk

Ⅰ. 서 론

1.1 연구배경

휴대용 모바일 기기의 발전과 위치 인식 기술의 발전으로 인하여 위치 기반 서비스에 대한 관심이 급격하게 증가하고 있다. 90년대 중반이후 휴대폰, PDA 또는 스마트폰 같은 모바일 기기들의 발전과 GPS나 RFID 같은 위치를 인식할 수 있는 기술의 발전으로 위치 기반 서비스 (LBS: Location Based Services)에 대한 관심이 집중되고 있다 [1, 2]. 위치 기반 서비스는 사용자의 위치 정보를 기반으로 그에 관련된 서비스를 제공하는 것으로서 사용자의 위치에 관련된 정보제공, 광고, 주문배달 서비스, 쿠폰 서비스, 긴급경보 서비스, 물류관제 시스템 등이 있다. 또한, 이동 중인 차량을 대상으로 무선 인터넷 서비스를 제공하는 텔레매틱스 같은 서비스와의 결합을 통해 새로운 시장을 생성하여 다양한 서비스를 가능하게 한다. 이런 다양한 서비스는 우리 실생활에 밀접한 부분들로서 실용화와 함께 엄청난 이익을 창출할 것이다. 현재 위치 기반 서비스의 수준은 미비하지만 현재 GPS 모듈을 포함하는 휴대폰의 생산과 수요가 증가하고 있을 뿐만 아니라 지도 데이터를 기반으로 하는 다양한 콘텐츠가 개발되고 있기 때문에 앞으로 성장 가능성이 크다.

대규모의 확장성 있는 위치 기반 서비스를 제공하기 위해 소프트웨어 측면에서는 각 객체들에 대한 정보를 빠르게 검색하는 색인 기법이 핵심 기술 중의 하나이다. 특히 위치기반 서비스의 주체가 되는 객체는 지속적으로 이동하기 때문에 객체의 위치 갱신 연산이 수시로 발생할 뿐만 아니라 그에 따른 질

Collection @ chungbuk

의 결과도 수시로 변경된다. 지속적인 객체의 위치 갱신 연산은 색인의 성능에 큰 영향을 미치고 이는 곧 서비스의 저하를 초래한다. 그러므로 이러한 이동 객체 환경을 고려한 효율적인 색인 기법이 필수적이다.

이동 객체 환경은 객체와 질의의 이동 여부에 따라 크게 세 가지로 구분된다. 첫째, 객체가 이동하고 질의가 고정된 환경이다. 어떤 사람이 콜택시 회사에 택시를 요청하는 환경을 예로 들면, 질의의 주체가 되는 사람은 고정되어 있고 질의의 대상이 되는 택시는 객체로서 항상 움직인다. 둘째, 객체가 고정되고, 질의가 이동하는 환경이다. 한 운전자가 주변의 저렴한 주유소를 검색하는 경우를 예로 들면, 질의의 주체가 되는 운전자는 이동을 하지만 질의의 대상이 되는 객체인 주유소는 고정되어 있다. 셋째, 객체와 질의가 모두 이동하는 환경이다. 이러한 세 가지 환경에 따른 다양한 색인 기법들이 현재까지 연구되었다.

이동 객체를 대상으로 하는 질의 처리 기법은 향상된 위치 기반 서비스를 제공하기 위해 기술적인 관점에서 매우 중요한 부분이다. 또한 객체의 이동 때문에 질의 결과가 지속적으로 변하기 때문에 연속 질의 처리 기법은 중요한 연구 분야로서 관심이 집중되고 있다. 위치 기반 서비스에서는 객체의 이동성으로 인해 동일한 질의를 특정 시간 동안 수행하는 연속 질의(continuous query)가 매우 중요하다. 연속 영역 질의(continuous range query)는 이동하는 객체를 대상으로 위치기반 서비스를 제공하기 위해 중요한 질의 형태중의 하나이다. 연속 영역 질의란 이동 중인 객체를 대상으로 특정 영역 내에 존재하는 객체를 지속적으로 감시하는 질의이다. 연속 영역 질의는 지능형 운송 시스템(intelligent transportation systems) 또는 센서 기반 모니터링

Collection @ chungbuk

(sensor-based monitoring)과 같은 응용에서 자주 사용되며 향후 텔레매틱스 분야에서도 아주 중요한 질의 유형이 될 것이다. 예를 들어, 교통사고 또는 정체를 발생시키는 사건들이 발생한다면 교통소통을 원활이 하기 위해 해당 구역 주변의 차량에게 해당 정보를 제공하거나 향상된 119 서비스로 화재나 폭발 사고 같은 위급 상황이 발생할 경우 구역별로 대처 정보를 모바일 기기를 통해 제공해 줄 수 있다 [3, 4].

이동 객체에 대한 연속 질의를 처리하기 위해 객체 색인을 이용하는 많은 연구들이 진행되어 왔다 [23, 24, 25, 26]. 객체 색인 기법은 연속 질의를 처리하기 위해 객체의 위치 변화에 따라 주기적으로 질의를 재실행 하여 질의 결과를 매번 계산을 해야만 한다. 그러나 변경된 객체의 개수와 무관하게 전체 색인을 대상으로 매번 질의를 처리하기 때문에 많은 연산 비용을 소모한다. 다시 말해서 객체의 이동에 영향을 받지 않는 질의들에 대해서도 질의 처리를 재 수행하기 때문에 질의 처리 성능이 크게 감소한다. 연속 질의를 효과적으로 처리하기 위해서는 객체의 위치 변화에 따른 질의 결과의 변경을 점진적으로 처리하는 질의 색인 기법을 이용해야 한다. 질의 색인 기법은 기존의 객체 색인과 달리 질의에 영향을 주는 영역을 색인으로 관리하기 때문에 객체의 갱신 과정에서 객체의 갱신이 질의 결과에 영향을 미치는지 여부를 빠르게 판단할 수 있다. 또한 질의 결과에 영향을 미치는 갱신에 한 해서만 해당 질의의 결과를 변경하는 연산을 수행하기 때문에 불필요한 연산 비용의 소모를 방지한다.

연속 질의 처리 기법에서 고려되어야 할 사항은 질의 색인의 크기와 검색 성능이다. 연속 질의는 주로 빠른 응답 시간을 요하는 응용에서 사용되기 때

Collection @ chungbuk

문에 메인 메모리 기반으로 수행된다. 그래서 메인 메모리 기반의 빠른 질의 처리를 수행할 수 있는 색인 구조가 요구된다. 하지만 메인 메모리는 저장 공간의 제한을 가지기 때문에 질의 처리 기법의 평가는 질의 처리 속도뿐 만이 아니라 저장 공간의 효율성 측면에서도 이루어진다.

최근 몇 년간 연속 영역 질의 처리를 위해 많은 연구가 진행되어 왔다. 질의의 이동성 고려 여부에 따라 연속 영역 질의 처리에 대한 기존 연구는 크게 두 가지로 분류할 수 있다. 첫 번째는 이동하는 객체에 대해 질의가 고정되어 있는 상황에서의 연속 영역 질의 처리[5, 6, 7, 8, 9]이고 두 번째는 질의 자체가 이동하는 것을 고려한 연속 영역 질의 처리[10, 11]이다. 본 논문은 질의 자체가 이동하는 않는 정적인 질의만을 고려하고 있다. 연속 영역 질의를 처리하기 위해서는 질의 색인 기법이 필요하다. 최근까지 연속 영역 질의를 처리하기 위한 질의 색인 기법에 관련된 많은 연구들이 진행되었다. 연속 영역 질의를 처리하기 위해 객체의 이동에 따라 질의 결과에 변화를 디스크 기반 질의 색인을 통해 수행할 경우 빈번한 I/O가 발생하며, 이로 인해 질의 처리에 많은 시간이 소요된다.

지금까지 연구된 연속 질의 기법은 크게 연속 영역(range) 질의와 연속 k-최근접(k-Nearest Neighbor) 질의로 분류한다. 연속 질의 처리를 효율적으로 수행하기 위해 질의 색인이 제안되었고, 그에 따른 다양한 질의 색인 기법을 활용한 질의 처리 기법들이 제안되었다. 연속 영역 질의를 위한 디스크 기반 색인 기법과 질의 처리 기법으로는 R-트리와 안정 영역(safe region) 개념을 이용한 Q-Index[8], 질의의 이동을 함께 고려한 SINA[10] 그리고 질의 영역을 구분하기 위해 color 개념을 도입한 cGridex[9]이 제안 되었다. 그리고 연속

Collection @ chungbuk

영역 질의 처리를 위한 메모리 기반 색인 기법으로는 셀 기반 색인인 CQI[5] 그리고 가상 셀 기반 색인인 VCR[6]와 CES[7]이 제안되었다.

연속 k-최근접 질의 처리를 위한 기법들도 다양하게 연구되었다[13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. 연속 k-최근접 질의 처리를 위한 디스크 기반의 질의 처리와 색인 기법으로 k개의 객체를 보장하는 영역을 계산하고 그 영역과 겹치는 셀들을 모두 탐색하는 SEA-CNN[13]이 제안되었고, 안전 영역을 활용한 기법으로서 TB-kNN[14]이 제안되었다. 메모리 기반의 기법으로는 질의 점을 둘러싸는 셀들을 사각형 단위로 한 겹씩 검사하는 YPK-CNN[15], 질의 점을 중심으로 각 방향의 셀들을 그룹화하고 우선순위 큐를 이용한 CPM[16], 그리고 주변 셀을 원단위로 선택하고 근접한 순서로 탐색하는 Circular-Trip[17]이 제안되었다.

메모리 기반의 기법과 디스크 기반의 기법에는 두 가지 큰 차이점이 있다. 메모리는 속도는 빠르지만 가격적인 측면이나 하드웨어 적인 측면에서 용량이 제한되어 있어서 색인의 유지비용을 최대한 줄이는 것이 중요하다. 디스크 기반의 색인 기법은 색인의 응답 속도를 빠르게 처리하기 위해 디스크의 I/O횟수를 줄이는 것이 중요하지만 메모리 기반의 색인구조에서는 더 이상 디스크를 이용하지 않기 때문에 불필요한 연산을 줄이는 것이 중요하다. 따라서 메모리 기반에서 고려해야 할 점은 색인의 저장 공간 비용과 불필요한 연산을 줄이는 것이다. 기존의 메모리 기반 질의 색인 기법들은 접근 방법에 따른 성능상의 한계를 지니고 있다. 첫 번째로 객체가 어떤 질의의 범위로부터 벗어났는지 아니면 어떤 질의의 범위로 새롭게 포함되었는지를 파악하기 위해 객체가 이전에 포함되었던 질의들과 현재 포함하는 질의들을 질의 식별자를 이

Collection @ chungbuk

용하여 비교한다. 이때, 객체의 식별자를 이용하여 비교 연산을 하여야하기 때문에 많은 연산 비용을 소요한다. 두 번째로 메모리 기반 색인 구조가 대부분 그리드 기반으로 구성되기 때문에 그리드에 포함된 질의를 유지하기 위해서는 다수의 그리드에 질의의 식별자를 중복적으로 유지하여 저장 공간의 낭비를 초래한다.

Collection @ chungbuk

1.2 연구방법 및 범위

본 논문에서는 연속 질의 처리를 효과적으로 수행하기 위한 메모리 기반의 질의 색인 기법을 제안하고 그에 따른 연속 영역 질의 처리 기법 및 연속 k-최근접 질의 처리 기법을 제안한다.

연속 질의 처리를 위해 기존에 제안된 메모리 기반 질의 색인 기법을 문제점을 획기적으로 개선하여 보다 적은 유지비용으로 보다 빠른 연산을 수행하는 질의 색인 구조를 제안한다. 제안하는 질의 색인 기법은 객체의 이동에 따른 질의 결과를 변화를 빠르게 처리하는데 초점을 두었으며 메모리 기반의 그리드 구조로 구성한다.

질의 색인을 기반으로 연속 질의를 처리하기 위한 연산 비용을 최소화하기 위해 제안하는 연속 질의 처리 기법은 비트 식별자(Bit Identifier)라는 개념을 사용한다. 비트 식별자는 각 질의에 부여된 식별자로 그리드 색인의 각 셀은 질의들의 겹침 정보를 반영하는 비트 열을 가지고 있다. 이러한 비트 열을 통해 객체가 어떤 질의에 포함되었는지 파악할 수 있으며 셀 사이의 비트 열을 비교하여 질의들 간의 포함관계를 파악할 수 있다. 이러한 처리 과정은 대량의 객체에 대해 질의를 처리할 경우 큰 성능 향상을 보이며 기존 메모리 기반 기법에 비해 연산 비용을 감소시킬 수 있다. 제안하는 연속 질의 처리 기법은 저장 공간과 비트 열의 길이를 효율적으로 관리하기 위해 셀들을 그룹 단위로 관리하고 각 그룹은 그룹에 포함된 질의들을 관리한다. 이로 인해, 질의 식별자들의 중복 저장을 최소화하여 색인의 유지비용을 감소시킨다.

연속 k-최근접 질의 처리를 효율적으로 처리하기 위해서 본 논문에서는 거

Collection @ chungbuk

리 관계 패턴을 제안한다. 거리 관계 패턴은 한 셀 내의 임의의 점으로부터 주변의 다른 셀들을 거리에 대한 오름차순으로 정렬하고 그 셀들을 그 점이 존재하는 셀을 기준으로 상대 좌표로 나타낸 셀 좌표 리스트들의 셋이다. 이 패턴은 동일한 크기의 셀로 구성된 그리드에서 유효하다. 그리드 색인에서 k-최근접 질의를 처리하기 위해 가장 많이 차지하는 연산 비용 중의 하나인 근접한 셀을 찾는 연산을 최소화하기 위해 거리 관계 패턴이 사용된다. 또한 거리 관계 패턴을 고려한 질의 색인구조와 k-최근접 질의 처리 기법을 제안한다. 또한 본 논문에서는 k-최근접 질의 처리 시 거리 값을 기준으로 정렬된 거리 관계 패턴의 상대좌표를 순차적으로 적용하여 근접한 셀을 알아내기 때문에 우선순위 큐를 이용하는데 드는 연산비용을 절약하였다. 다시 말해서 제안하는 질의 처리 기법은 연속 질의 처리를 위해 거리 관계 패턴 외 추가적인 자료구조를 유지하지 않기 때문에 저장 공간의 낭비를 방지한다. 제안하는 질의 처리 기법의 성능을 검증하기 위해 가장 최신의 기법인 CPM과 비교를 수행한다.

본 논문에서 제안하는 연속 질의 처리 기법의 처리 시스템의 구조를 그림 1.1에서 나타낸다. 연속 질의 처리 시스템은 객체 이동시 영향을 받는 질의를 찾아내어 질의 결과를 갱신한다. 시스템 구성은 메모리 기반의 질의 색인, 질의 색인을 통해 연속 질의 처리를 수행하기 위한 효율적인 k-최근접 질의와 영역 질의 알고리즘 그리고 k-최근접 질의 처리를 위해 사용되는 거리 관계 패턴으로 구성된다. 본 논문에 기술된 내용의 요약은 아래와 같다.

Collection @ chungbuk

- 비트 패턴을 활용한 질의 색인 기법 제안
- 비트 패턴 기반의 연속 영역 질의 처리 기법 제안
- 거리 관계 패턴을 제안
- 거리 관계 패턴 기반의 k-최근접 질의 처리 기법 제안
- 저장 공간 및 질의 처리 비용 분석
- 다양한 성능 비교를 통해 제안하는 알고리즘의 우수성을 증명

그림 1.1 연속 질의 처리 기법의 구성

Collection @ chungbuk

1.3 본 논문의 구성

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구에 대해서 기술한다. 관련 연구로서 디스크와 메모리 기반의 색인 및 질의 처리 기법으로 분류하고, 질의 처리 기법은 연속 영역 질의와 연속 k-최근접 질의 처리 기법으로 분류하여 소개한다. 3장에서는 본 논문에서 제안하는 연속 영역 질의 처리를 위한 색인 구조 및 질의 처리 기법과 연속 k-최근접 질의 처리 기법을 위한 색인 기법과 질의 처리 기법을 구체적으로 기술한다. 4장에서는 제안하는 기법의 우수성에 대한 분석적인 비교를 보여준다. 5장에서는 성능 평가를 통해 제안하는 기법의 우수성을 입증하고 마지막 6장에서는 본 논문의 결론을 정리 한다.

Collection @ chungbuk

Ⅱ. 관련 연구

공간 객체를 위한 색인 기법들과 질의 처리 기법들이 다양하게 제안되었다. 공간 객체위한 대표적인 색인으로서 최소 경계 사각형(MBR : Minimum Boundary Rectangle)을 이용한 R-tree[12]가 제안되었고 이어서 이를 개선한 R^*-tree[30], R^+-tree[31]가 제안되었다. 노드의 모양을 사각형 대신 원을 이용한 SS-tree [32] 그리고 사각형과 원의 정보를 모두 저장하여 그 교집합을 노드의 영역으로 정의한 SR-tree[33]는 *k*-최근접 질의 처리 성능을 향상시키기 위해 제안되었다. *k*-최근접 질의 처리 시 단말 노드 간에 이웃 정보를 유지하여 불필요한 중간 노드의 접근은 방지한 ISR-tree[39]도 제안되었다. 하지만 고차원 데이터를 처리함에 있어서 각각 겹침 영역의 증가와 노드 팬-아웃의 감소에 따른 질의 처리 성능의 저하를 초래하기 때문에 고차원 데이터에 대한 질의 처리 성능을 향상시키기 위해 벡터를 이용한 TV-tree[34], X-tree[35] 그리고 VA-tree[36]도 제안되었다. 현재 위치와 과거 위치에 대한 질의 처리와 저장 공간의 낭비를 줄이기 위해 HR-tree[37], HR^+-tree[38]가 제안되었다.

최근 객체의 이동성을 고려한 색인 기법들이 연구되었다. R-tree 기반의 공간색인 기법들은 질의 처리 비용이 적게 소모되지만 갱신으로 인한 구조적 변경으로 인한 비용이 크게 발생하기 때문에 이동 객체 색인 기법으로는 적합하지 않다. 그래서 객체와 MBR에 벡터정보를 기록하여 객체가 이동하더라도 색인을 갱신하지 않는 TPR-tree[26]가 제안되었다. 벡터 정보의 이용은 사장 공간의 증가를 초래하기 때문에 질의 처리 성능 저하의 요인이 된다. 이를 해결하기 위해 특정주기를 기준으로 색인을 분할함으로서 벡터 정보로 인한 사

Collection @ chungbuk

장공간의 확장을 방지하여 질의 처리 성능을 높인 B^x-tree[27]와 B^{dual}-tree[28]가 제안되었다. B^x-tree는 그리드 구조로 색인을 하기 때문에 객체 밀집도의 변화에 따른 고려를 하지 못했다. 객체의 밀집도를 고려하여 매 주기마다 밀집도에 따른 색인 공간은 분할하고 분할된 공간의 객체 밀집도에 따른 그리드 해상도를 고려한 B^x-tree를 구축하는 ST^2B-tree[29]가 제안되었다.

연속 질의 처리를 위해 제안된 기법들은 저장장치로서 디스크를 대상으로 하는 기법들과 메모리를 기반으로 하는 기법들로 구분된다. 연속 질의 처리 기법은 영역 질의와 *k*-최근접 질의를 대상으로 소개한다.

연속 영역 질의 처리를 위한 메모리 기반 기법은 셀 기반 색인(cell-based index)[5]과 다양한 크기의 가상 셀을 이용한 가상 셀 기반 색인(virtual cell-based index)[6, 7]을 이용한 질의 처리 기법이 제안되었다. 디스크 기반 처리 기법은 기존 R-tree와 안전 영역(safe region) 개념을 이용한 Q-Index[8], 질의의 이동성을 함께 고려한 SINA[10] 그리고 그리드 기반 색인 구조에 칼라(color)라는 개념을 이용하여 질의 영역들 사이의 포함관계를 표현하는 cGridex[9] 기법이 제안되었다.

연속 *k*-최근접 질의 처리를 위한 메모리 기반 기법은 이전 시간의 결과를 이용하여 *k*개의 객체를 보장하는 영역을 계산하고 그 영역과 겹치는 셀들을 모두 탐색하는 SEA-CNN[13]이 제안되었고, 안전 영역을 활용한 기법으로서 TB-*k*NN[14]이 제안되었다. 메모리 기반의 기법으로는 질의 점을 둘러싸는 셀들을 사각형 단위로 한 겹씩 검사하는 YPK-CNN[15], 질의 점을 중심으로 각 방향의 셀들을 그룹화하고 우선순위 큐를 이용한 CPM[16], 그리고 주변 셀을 원단위로 선택하고 근접한 순서로 탐색하는 Circular-Trip[17]이 제안되었다.

Collection @ chungbuk

2.1 디스크 기반 질의 색인 기법

2.1.1 Q-Index

이동 객체를 대상으로 하는 연속 영역 질의를 처리하기 위한 질의 색인 기법으로 S. Prabhakar는 Q-Index를 제안하였다[8]. 수없이 발생하는 이동 객체의 갱신정보의 처리량을 줄이기 위해 제안된 것으로 안전 영역(safe region) 개념을 도입하여 이를 해결하고 있다. 즉, 이동 객체가 안전 영역을 벗어 낫을 때만 갱신을 하게 하는 방식이다.

각 질의가 포함하는 질의 영역 MBR(Minimum Bound Rectangle)을 통해 R-트리를 구성하고, 객체의 갱신으로 인한 질의 결과의 변경을 비교하기 위한 연산을 감소시키기 위해 실제 객체가 어느 질의의 결과에도 영향을 미치지 않는 영역으로 안전 영역(safe region)을 사용한다. 이러한 안전 영역은 객체가 갱신되어도 질의 결과에 영향을 미치지 않는 영역으로 객체의 갱신 과정에서 연속 질의 시간을 감소시킬 수 있다. 그림 2.1은 Q-Index와 안전 영역을 나타낸 것이다. 이동 객체들은 자신의 안전 영역을 미리 계산하여 알고 있으며 객체의 갱신된 위치가 안전 영역을 벗어났을 때에만 위치 정보를 질의와 비교한다. 객체의 갱신으로 질의 결과를 재계산하면 질의 색인은 객체의 새로운 위치를 기준으로 새로운 안전 영역을 계산하며 안전 영역을 벗어나지 않는 한 어떠한 객체도 질의 결과를 재평가하지 않는다.

Q-Index에서 사용하는 안전 영역은 그림 2.1에서 보는 것과 같이 *SafeDist, SafeRect, SafeSphere*와 같이 세 가지 유형으로 계산할 수 있다. *SafeDist*는 객체의 위치를 기준으로 가장 가까운 질의까지의 거리를 나타내며 객체가 질

Collection @ chungbuk

의 결과에 영향을 미치지 않고 이동할 수 있는 거리를 나타낸다. *SafeSphere*는 *SafeDist*와 유사한 개념으로 객체가 이동할 수 있는 영역 안전 영역을 원 형태로 나타낸 것이다. *SafeRect*은 객체의 위치를 기준으로 질의 결과에 영향을 미치지 않는 영역을 MBR 형태로 계산한 것으로 *SafeSphere*, *SafeDist*에 비해 안전 영역을 계산하기 위해 많은 시간을 소요할 수 있다.

Q-Index는 안전 영역을 통해 객체의 갱신으로 인해 질의 결과가 변경되지는 여부를 판별하기 위한 연산을 감소시킨다. 그러나 객체의 이동 변화가 매우 빠르거나 특정 지역에 많은 질의가 포함되어 있을 경우 안전 영역을 계산하기 위해 많은 시간을 소요할 수 있으며 최악의 경우에는 안전 영역이 질의를 처리하는데 도움이 되지 않을 수도 있다.

그림 2.1 Q-Index의 안전 영역(safe-region)

Collection @ chungbuk

2.1.2 SINA (Scalable INcremental hash-based Algorithm)

M. F. Mokbel는 고정된 위치에서 질의를 수행하는 정적 연속 질의(Stationary continuous query) 뿐만 아니라 이동 연속 질의(moving continuous query)까지도 함께 처리가 가능한 기법으로 SINA을 제안하였다[10]. SINA는 질의 결과의 변경을 판별하기 위한 재평가 과정을 해싱(hashing), 무효화(invalidation), 조인(join)과 같은 3가지 과정을 통해 수행한다. 첫 번째로 해싱에서는 메모리 버켓 내에 들어오는 객체와 질의의 갱신정보를 서로 비교하여 정 갱신(positive update)을 판별하고, 두 번째로 무효화 단계에서는 메모리 버켓 상의 갱신정보와 디스크 상의 이전정보를 비교하여 부정갱신(negative update)을 판별한다. 마지막 단계로서 조인은 메모리 버켓 상의 갱신정보와 디스크 상의 정보들을 비교하여 정 갱신과 부정 갱신을 판별한다. 이런 과정을 통해 모든 정 갱신과 부정 갱신을 판별하고 그것으로 갱신을 처리한다.

연속 질의에서는 이전에는 질의 결과에 포함되어 있었지만 객체의 갱신으로 질의 결과에서 삭제되는 부정 갱신(negative update)과 이전 질의 결과에는 포함되어 있지 않았지만 객체의 갱신으로 질의 결과에 새로 포함되는 정 갱신(positive update)을 쉽게 판별해야 한다. SINA는 이러한 부정 갱신과 정 갱신을 효과적으로 판별할 수 있는 기법에 제시하였다. 그림2.2는 SINA에서 위치 변화에 따라 질의 결과의 변경을 판별하기 위한 재평가(re-evaluaiton) 과정을 나타낸 것이다.

SINA는 질의 결과의 변경을 판별하기 위한 재평가 과정을 해싱(hashing), 무효화(invalidation), 조인(join)과 같은 3가지 과정을 통해 수행한다. 해싱 과

Collection @ chungbuk

정에서는 메모리 내에서 이루어지며 버퍼에 새로 입력되어진 이동 객체와 이동 질의(moving query)들에 대해서 각 질의에 대한 정 갱신을 구한다. 이때, 두 개의 해시 테이블이 이용하는데 하나는 객체에 대한 해시 테이블이며 나머지 하나는 질의에 대한 해시 테이블이다. 해시 테이블은 그리드 기반 색인 구조로 구성되어 있으며 이동 객체의 새로운 위치가 삽입되면 객체의 위치가 포함되는 셀에 입력되고 이동 질의가 삽입되면 질의의 위치가 포함되는 모든 셀에 질의를 입력한다. 그림 2.3은 메모리 내에서 해시 구조를 나타낸 것이다. 객체의 새로운 위치가 입력될 때 객체를 위한 해시 테이블에 저장하고 질의를 위한 해시 테이블을 검색하여 어떤 질의에 객체가 포함되는지에 대한 판별한다. 새로운 이동 질의가 입력될 때에는 기존의 이미 입력된 객체와 비교를 수행하여 정 갱신을 판별한다. 이러한 과정을 통해 정 갱신에 대한 결과 집합은 메모리 내에서 생성되며 버퍼가 가득 차거나 지정된 시간이 지날 때까지 이와 같은 과정이 지속적으로 수행된다.

버퍼가 가득 차거나 지정된 시간이 지났을 때 버퍼는 플러시가 이루어지고 무효화 과정이 수행된다. 무효화는 이동 객체와 이동 질의에 대해서 부정 갱신을 계산하는 과정으로 이전 시점의 질의와 객체의 정보를 유지하는 객체 색인과 질의 색인을 이용한다. 그리드 색인 객체의 정보와 질의의 정보를 유지하는데 객체의 정보에서는 객체를 포함하는 질의들의 리스트가 그리고 질의의 정보에서는 에서는 질의가 포함했던 객체들의 리스트가 포함된다. 이 과정은 크게 두 단계로 이루어지는데, 첫 번째는 이동 객체에 대한 무효화이고, 두 번째는 이동 질의에 대한 무효화 이다. 이동 객체에 대한 무효화는 해싱 단계에서 이미 계산된 이동 질의들의 정 갱신 결과 집합에서 이전 시점에서도 객체

Collection @ chungbuk

그림 2.2 SINA의 질의 처리 과정

의 위치를 포함했던 이동 질의들에 대해서 그 객체에 대한 정 갱신을 무효화하고, 이동 객체가 이전 시점에 소속 되었던 셀을 벗어 낫을 경우 그 시점에 그 객체를 포함했던 모든 이동하지 않았던 질의들에 대해서 그 객체에 대한 부정 갱신을 갱신 응답 테이블에 삽입하는 과정을 수행한다. 이동 질의에 대한 무효화는 이동 질의가 이전시점에 포함했던 객체들 중에서 이전 시점에서는 겹쳤지만 현재는 겹치지 않는 셀에 포함되는 모든 객체들을 부정 갱신으로서 갱신 응답 테이블에 삽입하는 과정을 수행한다. 이런 과정을 통해서 조인 과정에서 셀 단위로 계산을 수행할 수 있게 하는데, 다시 말하면 한 셀에 관련된 계산을 다른 셀에 있는 내용을 참조하지 않고 그 셀에 관련된 정보로만 계산이 가능하게 한다는 것이다.

Collection @ chungbuk

그림 2.3 SINA의 해싱 단계

조인 단계에서는 셀 별로 정 갱신과 부정 갱신을 판별 하는데 메모리 버켓 상의 이동 객체와 디스크 상의 이동하지 않은 질의들 그리고 메모리 버켓 상의 이동 질의와 디스크 상의 이동하지 않은 객체들을 비교하는 두 단계로 수행을 한다. 첫 단계로서 메모리 버켓 상의 이동한 객체와 디스크 상의 이동하지 않은 질의사이의 조인을 수행하면 이동 객체가 다른 셀로부터 이동했으면 그 객체를 포함하는 모든 이동하지 않은 질의들은 정 갱신을 갱신 응답 테이블에 삽입하고, 한 셀 안에서 이동한 객체는 이전에 그 객체를 포함했던 질의들과 현재 새롭게 포함하는 질의 들을 비교하여 정 갱신과 부정 갱신을 판별한 뒤 갱신 응답 테이블에 삽입한다. 두 번째 단계로서 메모리 버켓 상의 이동한 질의와 디스크 상의 이동하지 않은 객체 사이의 조인을 수행하면 각 셀 안에서 이전에 포함했던 객체들과 현재 새롭게 포함하는 질의들을 비교하여

Collection @ chungbuk

정 갱신과 부정 갱신을 판별한 뒤 갱신 응답 테이블에 삽입한다.

세단계의 과정을 거치면서 많은 정 갱신과 부정 갱신의 내용들이 삽입되는데, 그중에서 객체와 질의의 정 갱신과 부정 갱신이 존재하면 두 갱신 정보를 삭제하여 중복적인 처리를 방지한다. 만들어진 정 갱신과 부정 갱신 정보는 질의에게 보내지며, 해당 질의는 변화된 정보만을 받아서 갱신을 처리 하게 되므로 빠르고 효율적인 처리를 수행한다.

Collection @ chungbuk

2.1.3 cGridex(Color Grid Index)

그리드 구조를 갖는 기존의 기법들이 이동 객체가 이동을 하였을 때 객체를 포함하는 질의들이 동일함에도 불구하고 이전에 포함했던 질의들과 현재 포함하는 질의들을 서로 비교를 해야만 한다는 문제점을 해결하기 위해 X. Wang은 cGridex를 제안하였다[9]. 질의들 간의 다양한 겹침 관계에 있는 모든 영역에 대해 칼라(color) 영역으로 재 정의를 하고, 칼라는 그 영역을 포함하는 모든 질의들을 관리하는 방식이다. 객체가 이동을 했더라도 같은 칼라 영역일 경우에 불필요한 갱신을 줄이고 있다. 두 칼라 값을 해시 키 값으로 하고 값으로는 미리 계산한 두 칼라 간의 정 갱신과 부정 갱신을 갖는 해시 테이블을 이용함으로서 객체가 다른 칼라 영역으로 이동을 하더라도 해시 테이블을 이용하여 보다 빠른 처리를 수행한다.

이 기법은 칼라 값을 추가적으로 이용하여 질의 집합을 유지하여 색인상의 중복적인 질의의 저장을 횟수를 감소시켰다. 칼라 값을 이용하여 이동한 객체의 위치가 여전이 같은 결과 집합 안에 있는지 없는지를 알 수 있는 방법을 제공하여 불필요한 연산 비용을 줄였다.

그리드 색인 구조를 이용하고 있고 기존의 셀 기반 색인과 매우 유사한 구조를 가지고 있지만 그리드 상의 셀에서 질의들의 정보가 아닌 질의들의 집합을 유지하는 칼라 값과 셀 안에서의 그 영역 정보를 가지고 있다. 그림 2.4는 cGridex의 칼라에 대해서 나타낸 것이다. 그림 2.4의 (a)는 cGridex에서 이동 객체의 갱신에 따라 질의의 결과를 재평가하는 과정을 나타낸 것이다. 객체의 갱신으로 그 위치가 변경되었을 때에는 그림 2.4의 (b)와 같이 셀의 칼라 정보를 비교한 후 어떤 칼라 값에 포함되는지 판별한다. 객체가 다른 위치로 이동

Collection @ chungbuk

그림 2.4 cGridex의 칼라

을 하였을 때 그림 2.4의 (c)와 같이 이동한 위치의 칼라 값이 동일하다면 비록 셀을 벗어났을 지라도 어떤 질의에도 영향을 미치지 않는다는 것을 쉽게 알 수 있으므로 불필요한 연산을 줄인다.

칼라 값은 구분할 수 있는 값들의 한 종류로서 질의들의 포함관계를 나타낸다. 포함하는 질의들이 같으면 동일한 칼라로 나타내고 그렇지 않을 경우 다른 칼라 값으로 부여한다. 질의들의 겹쳐지는 부분을 모두 다른 칼라로 보고 그 칼라들을 질의 집합과 함께 칼라 테이블(color table)에서 관리하는 것을 나타내고 있다. 이런 칼라 값과 질의 집합은 칼라 테이블에 저장되어 관리 된다. 그림 2.5는 그리드 상에서 질의들의 겹쳐지는 부분에 대한 칼라들과 질의 집합을 칼라 테이블에서 관리하는 것을 나타내고 있다.

또한 갱신 과정을 빠르게 수행하기 위한 별도의 칼라 별 해시 테이블을 제공하고 있다. 해시키로는 두 칼라 값이고 값으로는 두 칼라 간에 정 갱신을 해야 하는 질의 집합과 부정 갱신을 해야 하는 질의 집합을 유지한다. 이 해

Collection @ chungbuk

그림 2.5 cGridex의 칼라 테이블(color table)

시 테이블은 모든 칼라들의 관계를 관리하지 않고 인접하는 칼라들 간의 관계만 관리한다. 인접하는 칼라들이란 한 질의 내에 포함되는 칼라들이다. 칼라의 갱신은 질의가 삽입 또는 삭제 될 때 수행되어 지며, 칼라의 갱신이 수행되어 지면, 칼라 간의 관계를 나타내는 해시 테이블의 내용도 갱신되어 진다.

객체가 갱신하는 과정을 살펴보면 다음과 같다. 일단 객체가 이동을 하여 위치정보를 갱신하면, 그 위치 정보를 이용하여 그리드 상에서 어떤 셀에 포함되는지를 계산하고, 그 셀의 칼라들의 영역정보와 비교하여 어떤 칼라에 포함되는지를 알아낸다. 그리고 이 과정을 객체의 이전 위치정보를 이용하여 동일하게 수행하고, 객체의 이전 위치는 어떤 칼라에 포함이 되는지 알아낸다. 이전위치의 칼라와 현재 위치의 칼라를 비교하여 만약 같은 값이면 아무런 갱신도 수행하지 않고, 만약 다르면 갱신을 수행한다. 갱신을 수행하는 과정에서는 칼라간의 관계를 나타내는 해시 테이블을 이용하는데, 두 칼라를 해시키로

Collection @ chungbuk

하여 해시 테이블을 검색 했을 시 그 값이 존재하면 그 결과를 읽어서 정 갱신과 부정 갱신을 수행하고, 그렇지 않으면 칼라별로 칼라 테이블을 검색하여 질의 집합을 구한 후 각 질의들을 비교하여 정 갱신과 부정 갱신을 수행한다.

Collection @ chungbuk

2.2 메모리 기반의 질의 색인 기법

2.2.1 CQI (Cell based Query Indexing)

D. V. Kalashnikov는 이동 객체의 수많은 갱신을 처리하는데 디스크 기반 연속 영역 질의 처리 기법의 속도 면에서의 한계를 해결하기 위해 메인 메모리 기반의 연속 영역 질의 처리 기법을 제안하였다[5]. 디스크 기반의 색인 기법은 디스크의 I/O를 최소한으로 하는데 비중을 두지만 메인 메모리 기반의 색인 기법은 저장 공간과 연산 속도에 비중을 둔다. 이 기법은 이러한 메인 메모리상의 색인 기법의 특성에 맞게 그리드 구조를 이용한 연속 영역 질의 처리를 위한 색인 구조를 제안 한다.

이 색인 구조는 전체 영역이 n * n 인 정사각형의 셀들로 구성되어진 그리드이고 질의들의 정보를 유지하는 질의 배열(query array)이 존재한다. 그리드의 각 셀은 자신을 겹치는 질의들의 리스트를 유지하고, 그 리스트는 해당 셀을 완전히 포함하는 것과 부분적으로 포함하는 것으로 두개가지가 존재한다. 그림 2.6은 이러한 셀 기반의 색인 구조에 대해서 나타낸 것이다. 객체들이 새로운 위치 정보를 업데이트하면 해당위치 정보를 같은 크기의 정사각형 셀들로 이루어진 그리드 상에서 그리드 전체 크기와 셀의 크기 그리고 실제 포함하는 영역의 크기를 이용하여 어느 셀이 객체의 위치를 포함하는지 계산한다. 해당 위치를 포함하는 셀을 찾아서 그 셀을 겹치는 질의들을 유지하는 리스트를 이용하여 알아낸다. 완전 리스트(Full List)와 부분 리스트(part list)가 있는데, 완전 리스트에 있는 모든 질의들에는 이 객체가 포함이 되어야 하고, 부분 리스트가 유지하는 질의는 각각 객체의 위치 정보와 비교를 하여 어떤 질의에

Collection @ chungbuk

포함 되는지 계산한다.

객체가 이동을 하고 다시 갱신해야 할 위치 정보를 보내오면 새로운 위치를 포함하는 셀을 찾는다. 그리고 이전의 위치를 포함하는 셀과 같은 셀인지를 확인한다. 만약 같은 셀이고 그 셀은 완전 리스트에만 질의가 있고 부분 리스트에는 질의가 존재하지 않을 때, 다시 말해서 해당 셀을 포함하는 질의가 그 셀을 모두 완전히 포함한다면, 또는 완전 리스트에도 질의 가 존재하지 않을 때, 그 갱신 정보는 어떤 질의의 결과에도 영향을 미치지 않는 것이 확실하기 때문에 불필요한 연산으로서 수행을 하지 않는다. 만약 이전의 위치를 포함하는 셀과 같고, 그 셀에서 유지하는 부분 리스트에 질의가 있다면 완전 리스트에 있는 질의와는 비교하지 않고 부분 리스트에 있는 질의 하고만 비교를 하여 해당 위치 정보가 어떤 질의에 포함이 되는지 계산을 한다. 만약 이전의 위치를 포함하는 셀과 다른 셀에 포함되면 완전 리스트와 부분 리스트에서 유지하고 있는 모든 질의와 비교를 한다. 그리고 객체의 이전 위치를 포함하는 질의와 새로운 위치를 포함하는 질의를 비교하여 어떤 질의에서 벗어났고 어떤 질의로 새롭게 포함 되었는지를 비교하여 계산하다. 그리고 변경이 일어난 질의에 대해서 갱신을 해준다.

질의의 삽입 삭제 또한 간단하게 처리 된다. 새로운 질의가 삽입되면 해당 질의가 포함하는 모든 셀을 계산하고, 그중에서 질의가 완전히 포함하는 셀들은 완전 리스트에 그 질의의 포인터를 삽입하고, 부분적으로 포함하는 셀들은 부분리스트에 그 질의의 포인터를 삽입한다. 삭제 시에도 마찬가지로 영역을 계산하여 해당 셀들에 대해서 각 셀이 유지하는 완전 그리고 부분 리스트에서 그 질의를 삭제한다.

Collection @ chungbuk

이 기법은 셀을 완전히 포함하는 질의는 상관없지만 부분적으로 포함하는 질의에 관해서 객체 갱신 시 질의의 영역정보를 매번 query array에서 읽어와서 해당 위치정보와 비교를 해야 한다. 비록 이동 객체가 같은 셀 안에 여전이 존재한다고 해도 부분적으로 포함하는 질의에 대해서는 어쩔 수 없이 비교를 해야 한다. 셀의 크기를 어떻게 설정하는 가에 따라서 성능에 큰 영향을 미치게 되는데, 셀의 크기가 커질수록 부분적으로 겹치는 질의의 수가 많아지게 되어 색인의 검색 비용이 많이 들게 되며, 셀의 크기가 작아질수록 질의 식별자의 중복적인 저장이 많아지게 되어 저장 공간이 많이 소모되고, 삽입과 삭제 비용이 증가한다.

그림 2.6 셀 기반 질의 색인의 구조

Collection @ chungbuk

2.2.2 VCR(Virtual Construct Rectangle)

K. L. Wu은 기존의 셀 기반의 색인 기법의 단점을 보완하기 위한 가상 셀 기반 색인 기법을 제안하였다[6]. 셀 기반의 색인 기법은 이동객체가 이동했을 시 정보를 갱신하기 위해서 객체가 이전에 포함했던 질의들과 현재 포함하는 질의들을 서로 비교하는 방식을 취하고 있고 또한 같은 셀 안에서 이동한 경우에도 셀을 부분적으로 겹치는 질의들과 포함여부를 확인하기 위해서도 같은 계산을 수행한다. 이런 수행방식은 여전히 포함하는 질의에 대해서도 연산에서 제외시키지 않으므로 불필요한 연산을 수행하게 된다. 가상 셀 기반 기법은 동일한 크기의 셀이 아닌 다양한 크기의 셀들을 가상으로 만들고 질의를 다양한 가상 셀들로 분할하여 관리하여 이동 객체가 갱신을 했을 시 여전히 포함하고 있는 가상 셀들이 포함하고 있는 질의들에 대해서는 연산에서 제외시키는 점진적인(Incremental) 처리 방식으로 불필요한 연산을 줄였다.

가상 셀 기반 색인 기법은 그리드를 구성하는 셀의 사이즈는 모든 질의가 부분적으로 겹치지 않을 정도로 작게 지정한다. 그래서 셀의 단위로 질의의 크기단위를 지정하여 질의가 부분적으로 셀에 겹치는 경우가 발생하지 않는다. 이 기법은 여러 개의 셀들로 구성된 가상의 셀을 만들어서 질의를 유지한다. 이러한 가상의 셀을 여기서는 VCR이라고 한다. 이 VCR을 이용함으로서 유지비용을 효과적으로 개선했다. VCR은 정사각형으로 측면 길이는 2^n으로 만들어진다. 좌측 하단의 셀을 공유하여 관리하고 최대 크기를 제한하여 만들어지는 VCR의 개수를 제한한다. 그림 2.7은 최대 측면 길이(L)가 4일 때 좌측 하단의 셀을 기준으로 만들어 질수 있는 3개의 VCR들의 예를 나타낸다. (a), (b), (c)는 각각 측면 길이는 2^0, 2^1, 2^2가 된다.

Collection @ chungbuk

그림 2.7 VCR의 구성

질의는 여러 VCR들에 의해서 유지되고, 그 VCR들은 질의 삽입 시에 결정되어 지며, 질의의 식별자를 리스트로 유지한다. 이런 VCR은 정사각형으로 만들어 지고 그 크기는 2^n이다. 그림 2.8에서 보는 것과 같이 VCR로 분할하는 계산은 좌측 하단을 기준으로 시작하며 될수록 크게 지정한다.

VCR들을 효과적으로 관리하는 데는 두 가지의 조건이 추가된다. 첫 번째로 VCR의 저장 위치로서, 포함하는 셀들 중 좌측 하단의 좌표를 공유한다. 즉 좌측 하단의 좌표가 같은 모든 VCR은 같은 위치에서 관리가 됨으로서 VCR들의 정확한 저장 공간상의 위치를 찾을 수 있다. 다른 한 가지는 최대 사이즈가 미리 결정된다. 이는 만들어 질수 있는 VCR의 가지 수를 제한하고 검색 성능을 향상시킨다.

이동 객체가 위치 정보를 보내오면 해당 위치를 포함하는 셀을 계산하여 찾아내고, 해당 셀을 포함하는 모든 VCR들은 검색한다. 그림 2.9은 VCR을 검색하는 과정을 나타낸 것이다. 그 VCR들이 유지하고 있는 질의 리스트가 그 객체를 포함하는 질의들이 되는 것이다. 이동 객체가 새로운 위치 정보를 보내

Collection @ chungbuk

오면 이전 위치를 포함하는 VCR들과 현재 위치를 포함하는 VCR들을 비교하여 중복되는 VCR들은 이전 위치와 현재위치가 여전히 포함되어지는 질의 리스트를 가지고 있으므로 연산에서 제외시키는 방식으로 처리한다.

이 기법의 성능을 좌우하는 중요한 요소는 VCR의 최대 사이즈이다. 최대 사이즈가 커질수록 질의를 구성하는 VCR의 개수가 줄어들어 질의의 식별자를 저장 하는데 사용되는 저장 공간의 소모와 삽입과 삭제 시간이 줄어들지만 해당 위치정보를 포함할 가능성이 있는 VCR의 개수가 많아지기 때문에 검색 성능을 저하시키게 된다. 반대로 VCR의 최대 사이즈가 작아질수록 검색 성능은 좋아지지만 저장 공간의 소모는 증가한다.

그림 2.8 질의를 분할하는 다양한 크기의 VCR

Collection @ chungbuk

그림 2.9 VCR의 검색

Collection @ chungbuk

2.2.3 CES(Containment Encoded Squares)-Index

K. L. Wu는 기존에 제안한 가상 셀 기반의 색인 기법인 VCR-Index에서 검색 성능을 향상시키기 위해 가상 셀의 새로운 구성을 이용한 CES-Index를 제안했다[7]. 기존의 VCR은 특정 위치를 포함하는 가상 셀의 가지 수가 매우 많아서 검색 시 모든 가상 셀을 검색해야 하므로 성능의 저하를 가져온다. CES-Index는 만들어지는 가상 셀의 구조를 트리형태로 만듦으로서 특정 위치를 포함하는 가상 셀의 개수를 줄이고, 이로 인해 검색 성능의 향상을 이루었다. 여기서는 새로운 구성의 가상 셀을 CES 라고 한다.

CES가 VCR과 크게 다른 점은 가상 셀들 간에 트리 구조를 이루는 것이다. 그림 2.10에서 나타내고 있듯이, 색인은 그리드 구조이고 전체 영역은 CES의 최대 사이즈(L)와 동일한 크기의 셀로 이루어져 있다. CES들의 관리는 레벨 단위로 이루어지고 레벨0에서부터 시작하고, 레벨0은 최대 사이즈의 CES이고, 레벨1은 레벨0의 CES에서 4등분되어진 크기의 CES들로, 레벨2에서는 레벨1의 CES에서 4등분 되어진 크기의 CES들로 이루어져있고, 이런 방식으로 CES의 크기가 1이 될 때 까지 레벨을 만들어 CES을 관리한다. 그러면 각 레벨별로 CES의 개수는 4^i (i: 레벨)개가 되는 것을 알 수 있다. 지정된 CES의 최대 크기가 2^n 이라고 했을 때 레벨i의 CES의 크기는 2^{n-i} 라고 할 수 있고 이러한 레벨은 0에서 n까지 구성을 한다. 이렇게 구성한 각 레벨의 CES들은 z-ordering space-filling curve를 이용하여 각각 구분하기 위해 그리고 트리 구조를 위한 부모와 자식 간의 관계를 잘 표현하는 식별자를 지정한다. 여기서 부모와 자식관계란 레벨 k가 부모 레벨이라면 레벨 k+1은 자식이라고 할 수 있고 레벨 k의 한 CES은 레벨 k+1의 CES 4개를 자식으로 갖는다. 그림

Collection @ chungbuk

(R: 전체 그리드의 크기, L: CES의 최대 크기)
그림 2.10 CES의 구성

2.11은 CES 간의 트리구조와 레벨 별 CES의 포함관계를 나타낸다.

CES는 트리 구조를 이용함으로서 어떤 위치를 포함할 가능성이 있는 가상 셀의 개수를 줄였다. 다시 설명하면 그림 2.11에서 레벨2에서 어떤 위치정보가 식별자가 5인 CES에 포함된다면, 레벨1에서는 식별자가 1인 CES에 포함이 되고 그것은 레벨 0의 CES에 포함이 된다. 이 관계를 그림 2.12에서 보여주고 있다. 어떤 객체의 위치가 그림 2.12(a)에서 표현한 레벨2에서 식별자가 6인 CES에 있을 때, 그림 2.12(b)에서 표현한 레벨1에서는 식별자가 1인 CES에

그림 2.11 CES들이 나타내는 트리 구조

Collection @ chungbuk

포함이 되고, 그림 2.12(c)에서 표현한 레벨0에서의 CES에 포함이 된다. 그 외의 다른 CES에는 포함되지 않는 것을 알 수 있다.

객체가 위치 정보를 보내오면 처음 그리드 상에서 어떤 CES에 포함이 되는지 계산을 해야 한다. 가장 작은 크기의 CES(크기가 1인 CES)을 찾아내고 그 부모를 모두 찾아내는 것으로 해당 객체를 포함하는 CES을 모두 구할 수 있다. 그러면 해당 CES에서 유지하고 있는 질의가 그 객체를 결과로서 가지고 있는 질의라고 볼 수 있다. 이 기법에서는 효율적으로 부모 CES을 찾기 위해 CES에 식별자를 지정하고 해당 식별자를 이용하고 있다. 위의 그림 중 (b)에서 1은 가장 작은 CES이고 2는 그 부모 CES 그리고 3은 2의 부모 CES이다. VCR은 한 위치정보가 포함할 수 있는 가상 셀의 개수가 $(4L^2 - 1)/3$ 인데 비해 CES는 그 수가 log(L)+1로서 매우 적음을 쉽게 알 수 있다. (L은 가상 셀의 최대 사이즈를 나타낸다.)

질의 삽입 시 질의의 둘레에서 부터 양파의 껍질을 벗기듯이 CES를 계산한다. 우선 가장 작은 가상 셀의 크기인 $1(2^0)$로서 나눠야 하는 부분이 있는지 질의의 둘레를 보고 판단하고, 만약 나눠야 하는 부분이 있으면 나눠 낸다. 그

(a) 레벨 2 (b) 레벨 1 (c) 레벨 0

그림 2.12 CES들의 계층관계

Collection @ chungbuk

다음으로는 다음 크기인 $2(2^1)$로서 나눠야 하는 부분이 있으면 나눈다. 이런 식으로 CES의 최대 크기 까지 수행한다. 그림 2.13는 이런 과정을 이용하여 각각의 질의를 CES를 이용하여 분할 한 예를 나타낸다. 그림 2.13(a)는 나누기 전의 질의를 나타낸 것이고 그림 2.13(b)는 질의를 구성하는 CES을 나타내고 있다. 어둡게 표현된 부분은 다수의 질의가 같은 CES에 유지 되는 경우를 나타낸다. 이 기법 또한, VCR과 마찬가지로 가상 셀의 최대 크기의 지정이 성능을 크게 좌우한다.

(a) 질의 삽입 (b) 질의 분할

그림 2.13 CES에 의한 질의 분할

Collection @ chungbuk

2.3 기존 메모리 기반 질의 색인 기법의 문제점

연속 영역 질의를 효율적으로 처리하기 위해서는 객체가 위치 갱신을 할 때, 그 객체에 대해서 질의 결과에 영향을 받는 질의들을 빠르게 파악하는 것이 중요하다. 질의 결과에 대한 갱신이 필요한 질의를 알아내는 과정은 객체의 위치가 갱신될 때마다 수행되기 때문에 많은 연산 시간을 소요하기 때문이다. 기존의 연속 질의 처리 기법은 질의 색인을 통해 이동 객체가 이전에 포함되었던 질의 리스트들과 현재 포함되는 질의 리스트들을 계산하여 어떤 질의에서 벗어나고 어떤 질의에 새로 포함되는지를 계산한다. 이러한 수행 과정에서 질의 식별자들을 하나씩 비교해야하기 때문에 많은 연산을 소모한다. 뿐만 아니라, 실제 객체의 갱신으로 질의 결과에 영향을 미치지 않는 경우에도 매번 동일한 연산을 수행하기 때문에 많은 연산 비용을 소모한다.

그림 2.14은 이동 객체의 위치 갱신에 따른 질의와의 관계를 나타낸 것이다. 이동 객체 O1의 이전 위치를 포함하는 질의들은 Q1, Q2, Q3 그리고 Q4 이고, O1의 새로운 위치를 포함하는 질의들은 Q2, Q4, 그리고 Q7 이다. O1이 이동을 함으로써 Q1과 Q3에는 더 이상 포함되지 않고 Q7에는 새롭게 포함되며 Q2와 Q4에는 여전히 포함된다. 이때, O1에 대해서 삭제를 수행하는 질의는 Q1과 Q3이 되고, 삽입을 수행하는 질의는 Q7이 된다. 결과적으로 O1의 이동에 영향을 받는 질의는 Q1, Q3, Q7이 된다. 이러한 결과를 연산하기 위해 기존의 기법은 Q2, Q4에 대한 불필요한 비교 연산을 필요로 한다.

연속 영역 질의를 처리하기 위한 질의 색인은 질의를 관리하기 위해 질의 식별자를 유지하는데 일반적으로 메모리 기반 색인 구조는 그리드 구조를 사

Collection @ chungbuk

용하기 때문에 하나의 질의가 여러 셀에 포함되어 있을 경우 질의 식별자를 중복적으로 유지한다. CQI-Index의 경우에는 동일한 크기의 셀들로 구성되어진 그리드 구조를 이용하고 있다. 그리고 각 셀은 셀을 포함하는 질의들의 리스트를 유지하여 질의와 셀의 겹침 관계를 표현한다. VCR-Index와 CES-Index의 경우 그리드 구조를 이용하지만 동일한 크기의 셀이 아닌 다양한 크기를 나타내는 가상 셀을 이용하고 있다. 가상 셀도 질의와의 겹침 관계를 표현하기 위해 질의 리스트를 유지하지만 다양한 크기의 셀을 이용하여 질의 식별자의 중복 저장을 횟수를 감소시킨다. 그래도 여전히 각 셀에서 중복적으로 질의 식별자를 유지하는 횟수가 많기 때문에 저장 공간의 낭비를 초래한다.

그림 2.14 객체의 이동에 의해 영향을 받은 질의 연산

Collection @ chungbuk

2.4 디스크 기반 질의 색인을 활용한 *k*-최근접 질의 처리 기법

2.4.1 SEA-CNN

M. F. Mokbel는 고정된 위치에서 질의를 수행하는 정적 연속 *k*-최근접 질의(Stationary continuous *k*-NN query) 뿐만 아니라 이동 연속 *k*-최근접 질의(moving continuous *k*-NN query)까지도 함께 처리가 가능한 기법으로 SEA-CNN[13]을 제안하였다. SEA-CNN의 처리 방법은 SINA와 거의 유사하다. 하지만 *k*-최근접 질의 처리를 위한 추가적인 기법을 제공한다. SEA-CNN은 디스크 기반의 연속 *k*-최근접 질의 처리 기법으로서 다중의 연속질의를 수행함에 있어서 중복적인 디스크 접근을 방지한다. 초기 *k*-최근접 질의 결과를 구하는 알고리즘은 제안하지 않고 있지만 객체의 이동에 따라 수시로 변경되는 결과를 점진적으로 유지하는 효율적인 기법을 제안한다.

SEA-CNN은 과거 시점의 질의 결과를 이용하여 현재 시점의 결과를 보장하는 탐색 영역을 계산한다. 탐색 영역 내의 모든 객체를 대상으로 *k*-최근접 질의를 처리한다. 탐색 영역을 결정하는 방법은 몇 가지로 분류된다. 과거 결과 셋에 포함된 객체가 과거 질의 반경 보다 멀어진 경우 질의 점으로부터 가장 멀어진 객체까지의 거리를 반경으로 하는 원을 탐색 영역으로 지정한다. 과거 결과 셋에 포함된 객체 중 *k*번째 객체가 질의 점으로 더욱 근접해 진 경우 탐색 영역은 과거의 탐색 영역을 그대로 유지하고 질의를 수행한다. 질의가 이동할 경우 첫 번째 경우와 비슷한 경우로서 질의 점의 이동한 거리만큼 이전 탐색 반경의 길이를 연장하여 탐색 영역을 결정한다. 이러한 방식으로 *k*-최근접 객체를 보장하는 탐색 영역을 연산하여 점진적인 질의 처리를 수행한다.

Collection @ chungbuk

2.4.2 TB-*k*NN (Threshold based *k*-NN algorithm)

TB-*k*NN 기법은 연속 질의 처리 시 각 객체별 안전 영역을 결정하여 객체가 안전 영역을 벗어난 경우에 객체 이동에 따른 질의 처리를 수행하고 새로운 안전 영역을 계산하여 할당한다. 이 기법은 *k*-최근접 질의를 대상으로 안전 영역 개념을 사용한 최소의 기법으로서 서버와 객체간의 통신량을 감소시킨다[14].

안전 영역은 초기 *k*-최근접 질의 처리 후 결정된다. 결과 객체를 질의 점으로부터 근접한 순서로 정렬하고 그 순서를 기준으로 각 객체간의 중간 거리를 결정한다. 예를 들면 질의 점으로부터 3번째 근접한 객체는 각각 2번째 근접한 객체와 4번째 근접한 객체 사이의 중간 거리를 갖는다. 이 두 중간 거리 사이의 영역을 3번째 객체의 안전 영역으로 결정한다. 객체가 자신의 안전 영역을 벗어날 때까지 그 안전 영역을 유지한다. 각 질의는 안전 영역을 관리하기 위해 질의 결과인 *k* 번째 객체뿐만 아니라 *k*+1번째 객체의 정보도 관리한다.

Collection @ chungbuk

2.5 메모리 기반 질의 색인을 활용한 *k*-최근접 질의 처리 기법

2.5.1 YPK-CNN

YPK-CNN[15]은 영역 질의 기법을 *k*-최근접 질의 처리에 적용한 기법으로 *k*-최근접 객체들을 찾기 위해 셀 영역을 확장하면서 질의 처리를 수행한다. 주어진 질의 점을 포함하는 셀을 탐색하고, 만약 그 셀에 *k*개의 객체가 존재하지 않으면, 그 셀을 둘러싸는 주변의 셀들을 탐색한다. 검색 영역 확장을 통한 탐색은 *k*-최근접 객체를 보장하는 *k*개의 객체를 찾을 때까지 수행된다. 그림 2.15에서 YPK-CNN이 2-최근접 질의 처리 시 탐색하는 셀을 어두운 색으로 나타낸다. 연속 질의 처리 시 객체가 질의 점으로부터 *k*번째 객체까지의 거리 *k-dist*를 기준으로 밖에서 안으로 들어온 경우 질의의 결과 집합의 *k*-최근접 객체와 비교하여 결과 집합과 *k-dist*를 갱신하여 점진적인 처리를 수행한다. 하지만 객체가 *k-dist* 안에서 밖으로 벗어나거나 또는 질의 점이 이동한 경우 질의를 점진적인 처리를 하지 않고 질의를 재 수행한다. YPK-CNN은 사각 형을 기반으로 하고, 각 방향으로 동등하게 확장하므로 불필요한 셀을 많이 방문하여 처리 비용이 낭비된다. 또한 연속 질의 처리 시 *k-dist*를 기준으로 객체가 밖에서 안으로 들어올 경우에만 점진적인 처리를 수행하여 연속 질의 처리의 효율성이 낮다.

Collection @ chungbuk

(a) 2개의 객체 탐색 (b) 2-NN을 보장하는 객체 탐색

그림 2.15 YPK-CNN을 이용한 2-최근접 질의 처리

Collection @ chungbuk

2.5.2 CPM

CPM[16]은 YPK-CNN의 불필요한 셀 탐색 문제와 연속 질의의 비효율성의 문제를 해결하기 위해 제안되었다. CPM 기법은 가지치기(branch and bound) 알고리즘의 best-first 기법을 트리 구조가 아닌 그리드 구조에 적용한 기법이다. 이 기법은 우선순위 힙을 이용하여 항상 질의 점으로부터 가장 근접한 셀을 순서대로 찾는 것을 보장한다. 질의에 사용되는 메모리 사용량을 줄이기 위해 그리드 영역의 모든 셀을 힙에 삽입하지 않고, 질의 점를 중심으로 한 주변 셀들을 가상으로 그룹화 하여 삽입한다. 그룹은 트리 구조의 색인관점에서 중간 노드와 같다. 초기 힙에는 질의 점를 포함하는 셀과 그 주변 그룹들이 삽입된다. 그림 2.16에서 CPM이 2-최근접 질의 처리 시 탐색하는 셀을 어두운 색으로 나타낸다. 연속 질의 처리를 위해 셀 방문 리스트와 질의 처리를 위해 사용했던 힙을 유지한다. 객체가 *k-dist* 안으로 이동한 경우 YPK-CNN과 같은 방식으로 점진적인 질의처리를 수행한다. 객체가 *k-dist* 밖으로 이동한 경우, 셀 방문 리스트를 이용하여 근접한 셀을 순서대로 탐색한 후에 추가적인 셀 탐색이 필요할 경우 힙을 이용하여 점진적인 질의 처리를 수행한다. 셀 방문 리스트를 이용하기 때문에 근접한 셀을 알아내기 위해 소모되는 연산 비용을 감소시켰다. 만약 질의 점이 이동하면 셀 방문 리스트와 힙이 무효화 되므로 질의가 재 수행 된다. 그림 2.16에서 어두운 색으로 표시된 셀들은 모두 셀 방문 리스트에 유지되는 셀들이고, 그룹 *D2*, *R2*, *L3* 그리고 *U3*과 그 내부의 흰색으로 표시된 셀들이 정렬되어 힙에 유지되는 셀과 그룹들이다.

그림 1에서 비교하여 보여주는 것과 같이 CPM이 YPK-CNN의 불필요한 셀 탐색 문제를 해결하였다. 하지만 연속 질의 처리를 위해 셀 방문 리스트와

Collection @ chungbuk

힙을 질의마다 유지하여 저장 공간의 낭비를 초래한다. 이로 인한 저장 비용은 질의 점으로부터 탐색 범위와 질의 수에 비례하게 증가한다. 질의 처리 시 힙을 유지하기 위해 소모되는 비용 또한 무시할 수 없는 부분이다.

(a) 2개의 객체 탐색 (b) 2-NN을 보장하는 객체 탐색

그림 2.16 CPM을 이용한 2-최근접 질의 처리

Collection @ chungbuk

2.5.3 CircularTrip

CircularTrip[17]은 YPK-CNN과 비슷한 방식으로 탐색 영역을 확장한다. 그러나 YPK-CNN은 사각형 모양으로 탐색 영역을 확장하는 반면 CircluarTrip는 원모양으로 탐색 영역을 확장한다. CircularTrip는 힙 구조를 이용하여 참색 영역과 겹치는 셀들을 모두 탐색하지 않고 근접한 순서로 질의 처리를 수행한다. 그래서 불필요한 셀 탐색을 방지 하였다. 질의 처리는 두 가지 과정으로 수행된다. 첫 번째 단계로 CircularTrip는 질의 점을 포함하는 셀을 연산하고 질의 점과 그 셀의 변들과의 가장 근접한 거리를 반경으로 하는 원을 그린다. 그 후 질의 점을 포함하는 셀을 탐색하여 질의를 처리한다. 만약 질의 처리를 위해 추가적인 셀 탐색이 요구되면 처음 그림 원을 확장하여 새로운 원을 그린다. 확장된 원의 크기는 이전에 그려진 원에서 셀의 한 변의 크기만큼 반지름을 연장하여 결정한다. 확장된 원과 겹치는 셀을 근접한 순서대로 탐색하여 k 개의 객체를 탐색하고 추가적인 셀을 탐색해야 할 경우 동일한 방식으로 원을 확장하여 추가 탐색을 수행한다. 두 번째 단계로 질의 결과를 보장하는 결과를 연산하기 위해 k 번째 객체보다 근접한 셀을 모두 탐색한다. 이때 필요에 의해 탐색 영역을 확장을 수행한다.

그림 2.17은 CircularTrip을 이용한 2-최근접 질의 처리 과정을 나타낸다. C_1은 첫 번째로 만들어지는 원을 나타낸다. C_2는 두 번째로 만들어 지는 원으로서 그 반지름은 C_1의 것보다 셀의 한 변의 길이만큼 길다. 그림 2.17(a)의 경우 C_2와 겹지는 셀 탐색으로 2개의 객체 o_1, o_2가 탐색되었다. 그리고 질의 결과를 보장하기 위해서 추가적인 셀 탐색을 수행하고 탐색 영역을 확장한다. 그림 2.17(b)는 추가적인 셀 탐색으로 객체 o_3를 찾아내는 것을 나타낸다. 결

Collection @ chungbuk

과를 보장하기 위해서는 o_3 보다 질의 점에 근접한 모든 셀이 탐색 되어져야 한다. 그림 2.17에서 CircularTrip에 의해서 탐색된 셀을 어두운 색으로 표시하였다. CircularTrip은 주변 셀들을 질의 점과의 거리 순으로 정렬하기 위해 힙 구조를 사용한다. 그리고 CPM처럼 점진적인 연속 질의를 수행하기 위해 탐색했던 셀들을 순서대로 기록해 놓은 셀 방문 리스를 각 질의마다 유지한다. 하지만 질의마다 힙 구조를 유지하지 않기 때문에 저장비용은 CPM에 비해 적게 소모한다. 셀 방문 리스트는 CPM과 동일한 용도로 사용되고 추가적인 셀 탐색이 요구될 때는 k번째 객체를 기준으로 원을 그리고 그 원의 변과 겹치는 셀들부터 탐색함으로서 질의 처리를 수행한다.

(a) 2개의 객체 탐색 (b) 2-NN을 보장하는 객체 탐색

그림 2.17 CircularTrip을 이용한 2-최근접 질의 처리

Collection @ chungbuk

2.6 기존에 제안된 연속 *k*-최근접 질의 처리 기법의 문제점

CPM은 YPK-CNN의 불필요한 셀 탐색 문제를 해결하였다. 하지만 연속 질의 처리를 위해 셀 방문 리스트와 힙을 질의마다 유지하여 저장 공간의 낭비를 초래한다. 이로 인한 저장 비용은 질의 점으로부터 탐색 범위와 질의 수에 비례하게 증가한다. 질의 처리 시 힙을 유지하기 위해 소모되는 비용 또한 무시할 수 없는 부분이다. CircularTrip은 CPM에서 질의 마다 유지하는 힙을 제거하여 저장 공간의 효율성을 성취했지만 여전히 질의 처리 시에는 힙을 사용할뿐더러 셀 방문 리스트를 여전히 유지하기 때문에 질의 처리 비용과 저장 비용관점에서 모두 비효율 적이다.

본 논문에서는 이전 연구들의 문제점을 해결하고 보다 효율적인 연속 *k*-최근접 질의 처리 알고리즘을 제안하였다. 거리 관계 패턴을 제안하여 CPM에서 각 질의들이 유지하던 셀 방문 리스트와 힙을 대체하여 저장 공간의 소모를 방지하였고, 또한 힙을 이용하지 않기 때문에 힙을 유지하기 위한 연산 비용 소모를 방지하였다. 또한 연속질의 처리 시, 질의 점이 어느 정도 이동하더라도 연속질의를 점진 적으로 처리하는 알고리즘을 제안하였다.

Collection @ chungbuk

Ⅲ. 제안하는 연속 질의 색인 및 처리 기법

연속 질의 처리를 위한 기존 질의 색인 기법의 문제점을 해결하여 보다 효율적인 질의 색인을 만들기 위해 본 논문에서는 새로운 질의 색인 기법을 제안한다. 또한, 제안하는 질의 색인 기법을 이용한 연속 영역 질의 처리 기법을 제안한다. 먼저 제안하는 질의 색인 기법에 대해 설명하고 제안하는 질의 색인 기법을 이용한 연속 질의 처리 기법을 설명한다.

3.1 연속 질의 처리를 위한 색인 기법

3.1.1 전체 색인 구조

본 논문에서 제안하는 기법은 기존의 CES 기법과 셀을 그룹으로 관리한다는 면에서 비슷한 접근 방식을 가지고 있다. 그러나 색인 구조를 비교해 보면 비트 식별자를 이용한 질의 관리와 셀에서 유지하는 비트 열을 이용한 질의와의 포함관계를 표현하기 위한 구조로서 CES와는 매우 상이한 구조를 지니는 것을 알 수 있다. 그림 3.1은 연속 영역 질의 처리를 위해 본 논문에서 제안하는 기법의 전체적인 색인 구조를 표현하고 있다. 전체 그리드는 크기가 8×8인 64개의 그룹으로 구성되어져 있고, 6개의 질의(*Q1*~*Q6*)가 등록된 상태이다.

예들 들어, (6, 3) 위치에 있는 그룹으로 그룹과 겹치는 질의는 총 3개이다. 그중에서 *Q6*은 그룹을 완전히 포함하므로 *FL*에 등록되어 있다. 또한, *Q4*와 *Q5*는 그룹을 부분적으로 겹치므로 *PL*에 삽입된다. *PL*에 삽입 되어진 질의는 *QA*에 입력되고, 각 질의는 *QA* 내의 위치에 따른 비트 식별자를 갖는다. *Q4*

Collection @ chungbuk

의 비트 식별자는 0001이고, *Q5*의 질의 식별자는 0010이다. *PL*의 *GD*는 포함하는 셀을 관리하는데, *Q4*와 *Q5*에 포함되는 셀의 비트 열에 각각 질의의 비트 식별자 값을 더한다. 그래서 아무런 질의에도 포함되지 않는 셀의 비트 열은 0000, *Q4*에 또는 *Q5*에 포함되는 셀의 비트 열은 각각 0001과 0010이다. 그리고 *Q4*와 *Q5*에 모두 포함되는 셀의 비트 열은 0011 이다.

그림 3.1 제안하는 색인의 전체 구조

Collection @ chungbuk

3.1.2 질의 삽입

본 논문에서 제안하는 색인 구조에서 질의의 삽입은 삽입되는 질의의 영역과 겹치는 모든 그룹에 대해서 한 번씩 수행을 하고 질의가 그룹을 완전히 포함하는 경우와 부분적으로 겹치는 경우를 구분하여 완전히 겹치는 경우에는 *FL (Full List)* 그리고 부분적으로 겹치는 경우에는 *PL (Part List)*에 삽입한다. *PL*에 삽입이 되면 그룹 내의 그 질의와 겹치는 모든 셀의 비트 열에 질의의 비트 식별자를 더해서 그 셀들이 삽입된 질의에도 포함된다는 것을 나타낸다. 이렇게 *PL*에 삽입되는 질의 만 비트 식별자를 할당하고 *FL*에 삽입되는 질의는 그룹 내에 있는 모든 위치를 포함하기 때문에 이런 과정이 필요하지 않다.

그림 3.2은 질의 삽입을 처리하는 알고리즘이다. *G(q)*는 질의 *q*와 겹치는 그룹을 나타낸다. 질의영역과 겹치는 그룹이 질의에 의해서 완전히 포함되면 그룹의 *FL*에 질의를 삽입하기 위해 *InsertQueryIntoFL()* 수행하고, 그렇지 않으면 *PL*에 질의를 삽입하기 위해 *InsertQueryIntoPL()*을 수행한다.

그림 3.3은 질의를 *PL*에 삽입하는 알고리즘이다. *PL*은 *QA (Query Array)*, *QAH (Query Array Header)*, *GD (Grid)* 그리고 *NP (Next Node Pointer)*로 구성되어 있다. 그룹 *g*의 *PL*에 새로운 질의 *q*를 삽입할 때 질의 *q*가 그 *PL*에 삽입되는 첫 번째 질의라면 *PL* 노드 *Node_PL*을 생성하여 그룹 *g*의 *FL*에 삽입한다. 그리고 나서 *QA*에 질의를 삽입 할 공간이 있는지 확인하기 위해 만약 *QAH*의 비트 열의 각 자리의 수가 모두 1이면 다음 노드를 검사하는데 *QAH*의 비트 열이 각 자리의 수가 모두 1이 아닌 노드가 나올 때까지 한다. (줄 06)에서 이 과정으로 질의 *q*를 입력할 *PL* 노드를 찾는다. (줄 09)는

Collection @ chungbuk

*QA*상에서 *q*를 삽입할 위치를 찾는 부분으로서 *QHA*의 각 자리를 검사하여 값이 0인 부분을 찾는다. 만약 존재하는 모든 *PL* 노드의 *QA*에 질의 *q*를 삽입할 공간이 없으면 새로운 *PL*의 노드 *Node_PL*를 생성하고, 그 *PL* 노드에 질의 q를 삽입한 후 질의 *QAH* 값을 수정한다. (줄 13)는 *GD*상의 질의와 겹치는 모든 셀의 비트 열에 해당 질의의 비트 식별자를 더하는 연산을 한다.

그림 3.4는 한 그룹의 *PL*에 질의가 삽입되는 경우의 예를 나타낸다. 이 예는 한 그룹과 부분적으로 겹치는 *Q4*를 그 그룹의 *PL*에 삽입하는 것이고 (a)는 삽입 전의 상태, (b)는 삽입 후의 상태를 나타낸다. *Q4*가 입력되면 그룹의 *QA*에 어느 공간이 비었는지 알기 위해서 *QAH*를 보고 배열 *QA*의 두 번째 자리가 비었음을 알아내고 *Q4*를 배열 *QA*에 삽입한다. *Q4*가 두 번째 자리에 입력이 되었으므로 비트 식별자는 0010이 된다. 이 비트 식별자를 *QAH*의 비트 열인 1001에 더해주면 *QAH*는 1011이 된다. 그리고 질의의 비트 식별자를 *GD*상의 *Q4*와 겹치는 모든 셀의 비트 열에 각각 더해준다.

```
    Algorithm InsertQuery(lx, ly, hx, hy)
01    q = (lx, ly, hx, hy);
02    foreach g in ∀g ∈ (G(q)) {
03      if ( g is fully contained by q )
04        InsertQueryIntoFL(g, q);
05      else
06        InsertQueryIntoPL(g, q);
07    }
```

그림 3.2 질의 삽입 알고리즘(*InsertQuery*)

Collection @ chungbuk

```
Algorithm InsertQueryIntoPL(g, q);
  Node_PL_cur = g.PL;
  if (pl == NULL){
    create new Node_PL;
    insert the Node_PL into g.PL;
  }
  while ((Node_PL_cur.QAH == 2^(BIT_ID_SIZE)-1) and
          (Node_PL_cur.NP ≠ NULL))
    Node_PL_cur = Node_PL_cur.NP;
  if (Node_PL_cur ≠ NULL){
    i = 0;
    while (((Node_PL_cur.QAH >> i) % 2) ≠ 0) i++;
    Node_PL_cur.QAH += 2^i;
    Node_PL_cur.QA[i] = q;
    for each c in ∀c ∈ (C(q) in Node_PL_cur.GD) do c += 2^i;
  else{
    create new Node_PL;
    Node_PL.QAH += 2^0;
    Node_PL.QA[0] = q;
    for each c in ∀c ∈ (C(q) in Node_PL.GD) do c += 2^0;
    insert Node_PL into g.PL;
  }
```

그림 3.3 *PL*에 질의 삽입 알고리즘(*InsertQueryIntoPL*)

그림 3.4 *PL*에 질의 삽입

Collection @ chungbuk

3.1.3 질의 삭제

삭제는 질의의 삽입과 마찬 가지로 겹치는 모든 그룹에 대해서 한 번씩 수행하고 질의가 그룹과 완전히 겹치는지 아니면 부분적으로 겹치는지에 따라 *FL* 또는 *PL*에서 삭제를 수행한다. 삽입 과정은 *QA*에 질의 식별자를 기록하고 *QAH*에 비트 식별자 값을 더하는 과정을 했지만 삭제 과정에서는 *QAH*에서 비트 식별자 값을 삭제하는 과정만 수행한다. 이것은 *PL*과 *FL*에서 동일하게 수행되고 *PL*에서는 추가적으로 그룹 내에 질의가 포함하는 셀들의 비트 열에 질의의 비트 식별자 값을 삭제 해주는 과정을 수행한다. 그림 3.9은 질의 삭제를 처리하는 알고리즘이다. *G(q)*는 질의 *q*와 겹치는 그룹을 나타낸다. 질의와 겹치는 모든 그룹에 대해서 완전히 겹치면 그룹의 *FL*에서 질의를 삭제하기 위해 *DeleteQueryIntoFL()*을 수행하고, 그룹에 대해서 부분적으로 겹치면 그룹의 *PL*에서 질의를 삭제하기 위해 *DeleteQueryIntoPL()*을 수행한다.

그림 3.5은 그룹의 *PL*에서 질의를 삭제하는 알고리즘이다. (줄 05)에서 삭제할 질의 *q*의 식별자와 *PL*의 질의 식별자와 비교하여 같은 것을 찾는다. *QAH*에서 삭제하는 질의의 비트 식별자를 -2^i 연산 해준다. 질의 삭제 시 *GD* 상의 질의와 겹치는 모든 셀의 비트 열에 대해서 삭제하는 질의의 비트 식별자를 빼기 연산을 한다.

그림 3.7과 같이 *QA*상에서 *Q4*가 두 번째에 위치하므로 *Q4*의 비트 식별자는 0010이 된다. 이 비트 식별자를 *QAH*의 값인 1011에서 $-0010(-2^1)$ 연산을 하면 1001이 된다. 그리고 *RG*에서 *Q4*와 겹치는 모든 셀의 비트 열에서 *Q4*의 비트 식별자인 0010을 $-0010(-2^1)$ 연산을 한다. 결과는 그림 3.11의 (b)에서 나타내고 있다.

Collection @ chungbuk

```
Algorithm DeleteQuery(q)
  for each g in ∀g ∈ (G(q)) {
    if ( g is fully contained by q )
      DeleteQueryIntoFL(g, q);
    else
      DeleteQueryIntoPL(g, q);
  }
```

그림 3.5 질의 삭제 알고리즘(DeleteQuery)

```
Algorithm DeleteQueryFromPL(g, q)
  Node_PLcur = g.PL;
  while (pl ≠ NULL ){
    for (i=0 ; i<BIT_ID_SIZE ; i++){
      if (((Node_PLcur.QAH >> i) % 2) ≠ 0) continue;
      if (Node_PLcur.QA[i] == q){
        Node_PLcur.QAH -= 2^i;
        for each c in ∀c ∈ (C(q) in Node_PLcur.GD) do c -= 2^i;
        exit;
      }
    }
    Node_PLcur = Node_PLcur.NP;
  }
```

그림 3.6 PL에서 질의 삭제 알고리즘(DeleteQueryFromPL)

(a) Q4 삭제 전 (b) Q4 삭제 후

그림 3.7 PL에서의 질의 삭제

Collection @ chungbuk

3.2 연속 영역 질의 처리 기법

연속 영역 질의 처리는 두 가지로 분류되어 진행이 된다. 하나는 이동 객체가 그룹 내에서 이동을 한 경우와 나머지 한 가지는 그룹 밖으로 이동을 한 경우이다. 이동 객체의 그룹 내에서의 이동은 셀 간의 비트 열을 이용한 연산으로 결과 집합의 갱신이 필요한 질의를 파악한다. 그림 3.8에서 $G_{old}(O)$은 객체 O의 이전위치를 포함하는 그룹, $G_{new}(O)$은 객체 O의 새로운 위치를 포함하는 그룹, $C_{old}(O)$은 객체 O의 이전위치를 포함하는 셀의 비트 열, $C_{new}(O)$은 객체 O의 새로운 위치를 포함하는 셀의 비트 열, $OL(Q)$는 질의 Q의 객체 리스트, $G.FL$은 그룹 G의 FL (Full List), $G.PL$은 그룹 G의 PL (Part List) 그리고 $QL(b)$는 그룹 안에서 비트 열 b가 나타내는 질의들의 리스트를 나타낸다.

그림 3.8은 이동객체가 새로운 위치 값을 보내왔을 때 연속 영역 질의가 처리하는 알고리즘이다. (줄 01)에서 이동 객체의 새로운 위치를 포함하는 그룹과 이전 위치를 포함하는 그룹을 계산한다. (줄 02)에서 만약 두 그룹 g_{old}와 g_{new}가 같으면 이동객체의 이전 위치를 포함하는 셀의 비트 열(c_{old})과 새로운 위치를 포함하는 셀의 비트 열(c_{new})을 구한다. c_{old}과 c_{new} 두 값이 같으면 이동객체가 셀 안에서 이동을 한 것이기 때문에 어떤 질의에도 영향을 미치지 않는다. 만약 그 두 값이 같지 않으면, 객체 O가 어떤 질의에 영향을 미쳤는지 계산을 한다. c_{old}과 c_{new} 의 비트 열의 각 자리를 비교하여 c_{old}가 0이고 c_{new}가 1이면 해당 자리 비트를 식별자로 하는 질의의 결과 집합에 객체 O를 삽입하고, c_{old}가 1이고 c_{new}가 0이면 해당 자리 비트를 식별자로 하는 질의의

Collection @ chungbuk

결과 집합으로 부터 객체 O를 삭제 한다. (줄04~16)에서는 위의 처리를 PL에 있는 모든 노드에 대해서 처리하게 한다. (줄 18~23)에서 만약 두 그룹 g_{old}와 g_{new}가 다르면 객체를 현재 포함하는 모든 질의들의 결과 집합에 객체를 삽입하고, 이전에 포함했던 모든 질의들의 결과 집합으로부터 객체를 삭제하는 과정을 수행하여 질의들의 결과 집합을 갱신한다.

```
    Algorithm CQProcessing(Object O)
01    g_old = G_old(O); g_new = G_new(O);
02    if (g_old == g_new ){  // 이동 객체가 그룹 안에서 이동을 한 경우의 처리
03      Node_PL_cur = g.PL;
04      while (Node_PL_cur ≠ NULL ){
05        c_old = C_old(O) in Node_PL_cur;
06        c_new = C_new(O) in Node_PL_cur;
07        if (c_old ≠ c_new){
08          for (i=0 ; i<BIT_ID_SIZE ; i++){
09            if ((((c_old >> i) % 2) == 0) && (((c_new >> i) % 2) == 1))
10              insert O into Node_PL_cur.QA[i];
11            else if ((((c_old >> i) % 2) == 1) &&
                          (((c_new >> i) % 2) == 0))
12              delete O from Node_PL_cur.QA[i];
13          }
14        }
15        Node_PL_cur = Node_PL_cur.NP;
16      }
17    }
18    else{  // 이동 객체가 다른 그룹으로 이동을 한 경우의 처리
19      c_old = C_old(O);
20      c_new = C_new(O);
21      insert O into OL(q), ∀q ∈ (g_new.FL ∪  QL(c_new));
22      remove O from OL(q), ∀q ∈ (g_old.FL ∪  QL(c_old));
23    }
```

그림 3.8 연속 영역 질의 처리 알고리즘(*CQProcessing*)

Collection @ chungbuk

3.3 연속 *k*-최근접 질의 처리 기법

본 논문에서 제안하는 색인 기법은 동일한 크기를 가지는 셀 간의 거리 관계 패턴을 이용하여 질의 처리 시 적은 비용으로 근접한 셀들을 알아낸다. 그리고 한 점과 셀과의 거리 측정은 CPM과 동일한 방식을 이용한다. 표 3.1은 본 논문에서 사용하는 기호들을 나타낸다.

표 3.1 기호 설명

기호	설명
PC(p)	점 *p*가 포함되는 셀 내에서의 상대 좌표
CLA(p)	점 *p*로부터 셀들을 근접한 순으로 절대좌표로 정렬하여 표현한 리스트
CLR(p)	점 *p*로부터 셀들을 근접한 순으로 상대좌표로 정렬하여 표현한 리스트
N_{clr}	한 셀 내에서 *CLR*의 기준이 되는 점들의 개수
N_{sort}	한 *CLR*에서 유지하는 상대 좌표의 개수
k-dist	질의 결과 셋에서 *k*번째 객체와 질의 점과의 거리
CP(v)	영역 *v*의 중심점
MINDIST()	인자로 입력된 두 점 또는 점과 셀과의 최소 거리
MAXDIST()	인자로 입력된 점과 셀간의 최대 거리

Collection @ chungbuk

3.3.1 거리 관계 패턴

거리 관계 패턴은 한 셀 내의 임의의 점으로부터 다른 셀들을 거리에 대한 오름차순으로 정렬하고 그 셀들을 상대 좌표로 나타낸 셀 좌표 리스트이다. 거리 관계 패턴은 동일한 크기의 셀들이 동일한 간격으로 구성된 형태의 색인 구조에서 k-최근접 질의 처리를 위해 사용될 수 있다.

정의 1. 그리드 상의 위치가 다른 두 점 p_n, p_m 에 대해서 만약 $PC(p_n) = PC(p_m)$ 이면, $CLA(p_n) \neq CLA(p_m)$이지만 $CLR(p_n) = CLR(p_m)$이다.

정의 1은 다른 두 점이 셀 내에서의 상대 좌표가 동일하다면 주변 셀과의 상대적인 거리 관계도 동일하다는 것을 보장한다. 그림 2는 정의 1의 예를 보여준다. 그림 3.9(a)는 다른 셀에 존재하지만 셀 안에서의 위치는 동일한 임의의 두 점 p_1과 p_2를 나타내고, 그림 3.9(b)는 각 두 점의 CLA와 CLR을 나타낸다. 두 점이 존재하는 셀이 다르기 때문에 $CLA(p_1)$과 $CLA(p_2)$는 다르다. 하지만 $CLR(p_1)$과 $CLR(p_2)$는 동일한 것을 확인할 수 있다.

CLR은 CLA와는 다르게 셀 내에서의 상대좌표만 동일하다면 모든 셀에서 동일하게 만들어진다. 이러한 특성을 이용하여 셀 내의 모든 점에 대해서 CLR을 유지한다면 별도의 연산 없이 근접한 셀들을 차례로 접근할 수 있다. 하지만 셀 내의 모든 점에 대해서 CLR을 유지한다는 것은 거의 불가능 하다. 공간에서 점의 개수는 매우 많기 때문이다. 그래서 셀 내에서 대표적으로 의미 있는 점들을 추출하여 그 점들을 기준으로 CLR을 유지한다. 이 CLR들의 집합을 본 논문에서는 거리 관계 패턴으로 정의한다. 정의 2에서 그 거리 관계 패턴을 설명한다.

Collection @ chungbuk

정의 2. N_{clr}은 거리 관계 패턴이 가지는 *CLR*의 개수이다. 셀 영역을 가상으로 N_{clr} 만큼 분할하고, 그 분할된 작은 셀을 *VDC (Virtually Divided Cell)*라고 한다. 한 *VDC v*의 중심점은 *CP(v)*로 표현하고, *CLR(CP(v))*는 그 v 영역을 대표하는 *CLR*이다.

한 *CLR(CP(v))*가 나타내는 *CLR clr*은 배열구조를 가지며 *clr[n]*은 점 *CP(v)*로부터 *n*번째 근접한 셀의 상대 좌표를 나타낸다. 정의 3에서 그 관계를 설명한다.

Closest Cell list of p_1

CLA(p_1)	CLR(p_1)
C(2,4)	(0,0)
C(1,4)	(+1,0)
C(2,5)	(0,+1)
C(1,5)	(-1,+1)
C(2,3)	(0,-1)
…	…

Closest Cell list of p_2

CLA(p_2)	CLR(p_2)
C(6,3)	(0,0)
C(5,3)	(+1,0)
C(6,4)	(0,+1)
C(5,4)	(-1,+1)
C(6,2)	(0,-1)
…	…

(a) 임의의 두 점 위치 표현 (b)각각 두 점에 대한 *CLA*와 *CLR* 표현

그림 3.9 *CLA*와 *CLR*의 차이

Collection @ chungbuk

정의 3. 주어진 *VDC v*에 대해서 *CLR*은 식 3.1과 같은 속성을 갖는다.

$$MINDIST(cp, clr[i]) \le MINDIST(cp, clr[i+1])$$
$$\text{단}, cp = CP(v), clr = CLR(cp) \qquad (3.1)$$

그림 3은 N_{clr}의 값에 따라 셀 내의 분할된 영역과 만들어진 *CLR*의 정확성을 보여준다. 그림 3.10(a)는 N_{clr}이 1이기 때문에 셀의 중심점이 그 *CLR*의 기준점이 되고 그 영역을 대표한다. 그림 3.10(b)와 3.10(c)는 N_{clr}이 각각 4와 36이기 때문에 영역을 4개와 36개로 분할한다. 그리고 분할된 영역의 중심점이 각 *CLR*의 기준점이 된다. 분할된 영역별로 *CLR*을 만들고 그 영역에 포함되는 점을 기준으로 근접한 셀들을 찾을 때는 그 *CLR*을 따른다.

거리 관계 패턴의 최대 오차는 N_{clr}에 의해 분할된 한 영역 *VDC v*내의 모든 점들과 중심점인 *CP(v)*로부터 다른 셀들까지의 거리 측정 시 발생할 수 있는 최대 거리 차이이다. 그 오차를 정의 4에서 설명한다.

(a) N_{clr} = 1 (b) N_{clr} = 4 (c) N_{clr} = 36

그림 3.10 N_{clr}에 따른 거리 관계 패턴

Collection @ chungbuk

정의 4. 그리드 내의 셀들의 집합을 $C=\{c_1, c_2, ..., c_n\}$로 나타내고, 주어진 한 영역 *VDC* v 내의 점들의 집합을 $P_v=\{p_1, p_2, \cdots, p_n\}$로 나타낸다. 최대 오차 E는 식 3.2와 같다.

$$E(C,v,P_v) = \max\{|\,MINDIST(CP(v),c_i) - MINDIST(p_i,c_i)\,|\}$$
$$\text{단}, c_i \in C, p_i \in P_v \qquad (3.2)$$

정의 5. 2차원 평면에서 세 개의 점 p1, p2 그리고 p3가 있을 때, 삼각 부등식에 의해 $|MINDIST(p1,p3) - MINDIST(p2,p3)| \leq MINDIST(p1,p2)$은 보장된다. 만약 임의의 사각형 B가 있을 때, $|MINDIST(p1,B) - MINDIST(p2,B)| \leq MINDIST(p1,p2)$을 만족한다.

정의 5는 한 *VDC* v 에 존재하는 두 점 p_i와 p_j가 있을 때, $E \leq max\{MINDIST(pi, pj)\}$를 보장한다. 그래서 거리 관계 패턴의 최대 오차는 v에 포함된 점들 중 $CP(v)$로부터 가장 멀리 떨어진 점까지의 거리이다. a가 한 셀의 측면 길이라고 할 때, 최대 오차 E는 식 3.3과 같고, 그림 3.11은 N_{clr}이 4일 때, 거리 관계 패턴의 최대 오차 E를 보여준다. 어둡게 표시된 *VDC* v에서 $CP(v)$와 모서리의 한 점 p까지 거리가 최대 오차 E를 나타낸다.

$$E = \alpha\sqrt{2} / (2 \times \sqrt{\mathrm{n_{clr}}}) \qquad (3.3)$$

Collection @ chungbuk

그림 3.11 최대 오차 E 계산

이러한 오차 때문에 거리 관계 패턴은 한 VDC내의 모든 점들에 대해서 만족되지 않고 일부의 점들에 대해서만 만족된다. 이러한 관계를 정의 6에서 설명한다.

정의 6. 주어진 VDC v 내의 점들의 집합을 $P_v=\{p_1, p_2, \cdots, p_n\}$로 나타낸다. $cp = CP(v)$, $clr = CLR(cp)$라고 가정한다. 조건 $MINDIST(p_i, clr[k]) + MINDIST(p_i, cp) < MINDIST(cp, clr[k])$ 을 만족하는 점 p_i에 대해서는 조건 $MINDIST(p_i, clr[k]) < MINDIST(cp, clr[k + \beta])$이 성립되어 거리 관계 패턴이 유효하지만, 그렇지 않은 점 p_i에 대해서는 거리 관계 패턴이 유효하지 않다. 단, $\beta \geq 0$ 를 만족하는 정수이다.

Collection @ chungbuk

거리 관계 패턴을 기반으로 영역 v에 존재하는 임의 점 p_i로부터 근접한 셀들을 순차적으로 접근할 때, p_i와 $CP(v)$의 거리 차이로 인해 정확성을 보장하지 못할 수도 있다. 거리 오차는 $MINDIST(p_i, CP(v))$보다 작으므로 원하는 검색 반경을 D_{want}라고 할 때, 실제 검색 반경 D_{search}는 식 3.4와 같다.

$$D_{search} = D_{want} + MINDIST(p_i, CP(v)) \tag{3.4}$$

거리 관계 패턴은 두 개의 인자를 이용하여 생성된다. 하나는 정렬할 셀의 개수 N_{sort}이고, 나머지 하나는 CLR의 기준이 되는 점들의 개수 N_{clr}이다. N_{sort}와 N_{clr}은 거리 관계 패턴이 차지하는 저장 공간에 영향을 미친다. 또한 N_{clr}은 거리 관계 패턴의 정확성에 영향을 미친다. 거리 관계 패턴의 저장 비용은 정의 7에서 설명한다.

정의 7. 거리 관계 패턴의 저장 비용은 N_{sort}와 N_{clr}에 비례한다. 한 좌표를 표현하기 위한 저장 비용을 S_{crd}로 나타낼 때, 거리 관계 패턴을 위한 저장 비용 S_{pat}는 식 3.5와 같다.

$$S_{pat} = S_{crd} \times N_{sort} \times N_{clr} \tag{3.5}$$

거리 관계 패턴을 이용하면 적은 연산 비용으로 근접한 셀들을 순서대로 접근할 수 있다. 그리고 거리 관계 패턴의 오차를 줄이기 위해 N_{clr}을 크게 지정할 수 있지만 N_{clr}의 증가는 저장 공간의 소모를 초래하기 때문에 적절한 값을 결정해야 한다.

Collection @ chungbuk

3.3.2 *k*-최근접 질의 처리

본 논문에서 제안하는 패턴 기반의 *k*-최근접 질의 처리 기법은 거리 관계 패턴을 이용한다. 거리 관계 패턴은 *CLR*의 집합으로서 해시 함수를 통해 *CLR*을 할당한다. 할당 받은 *CLR*을 이용하여 *k*-최근접 질의 처리를 수행한다. 표 3.2는 질의 처리 알고리즘을 표현하기 위한 기호들을 나타낸다.

표 3.2 알고리즘 표현을 위한 기호

기호	설명
a	한 셀의 측면 길이
k	최근접 객체의 수
HCLR(q)	질의 *q*의 좌표를 포함하는 *VDC*를 *v*라고 할 때, *CLR(CP(v))*
QC(q)	질의 *q*를 포함하는 셀
AC(c, rc)	셀 *c*를 기준으로 상대 좌표 *rc*에 존재하는 셀
OL(c)	셀 *c*의 객체 리스트
QL(c)	셀 *c*의 질의 리스트
q.x, q.y	공간 상에서 질의 *q*의 *x*축의 좌표와 *y*축의 좌표
q.result_set	질의 *q*의 결과 집합
$q.p_{clr}$	질의 *q*가 탐색한 *CLR*상의 위치를 나타내는 값
q.k-dist	질의 *q*와 *q.result_set*내의 *k*번째 근접한 객체간 거리
q.qid	질의 *q*의 식별자
clr[i]	임의 *CLR clr*의 *i*번째 상대좌표
clr[i].x	임의 *CLR clr*의 *i*번째 *x*축의 상대좌표
clr[i].y	임의 *CLR clr*의 *i*번째 *y*축의 상대좌표
clr[i].dist	질의 *q*의 좌표를 포함하는 *VDC*를 *v*라고 할 때, *DIST(CP(v), clr[i])*
clr[i].msxdist	질의 *q*의 좌표를 포함하는 *VDC*를 *v*라고 할 때, *MAXDIST(CP(v), clr[i])*

Collection @ chungbuk

질의 처리는 3가지 단계로 수행된다. 첫 번째로 N_{sort}와 N_{clr}을 인자로 하여 거리 관계 패턴을 생성한다. 거리 관계 패턴은 그리드를 구성하는 셀의 모양과 배치를 기반으로 해서 만들어지기 때문에 한번 생성하면 영구적으로 사용 가능하다. 또한 거리 관계 패턴은 *CLR*들을 관리하고 좌표를 해시 키로 사용한다. 두 번째로 질의를 생성하고 *CLR*을 할당 받는다. *k*-최근접 질의는 질의 점, 검색할 객체의 개수, 결과 객체, 할당 받은 *CLR* 정보를 갖는다. 거리 관계 패턴으로부터 질의 점을 키로 적절한 *CLR*을 할당 받는다. 세 번째로 할당 받은 *CLR*을 이용하여 *k*-최근접 질의를 처리한다. CLR의 상대 좌표를 순서대로 참고 하여 *k*개의 객체가 검색될 때까지 근접한 셀을 순서대로 탐색한다. 우선 질의 점이 존재하는 셀을 탐색하고, 발견된 객체가 *k*개 미만이면, CLR의 첫 번째 상대 좌표로 다음 셀을 알아내고 탐색한다. 발견된 객체가 *k*개 미만이면, 다시 *CLR*의 두 번째 상대 좌표로 다음 셀을 알아내고 탐색한다. 이러한 방식으로 *k*객체를 찾을 때까지 셀을 탐색한다. 이때 질의 점에서 *k*번째 객체까지의 거리를 *k-dist*라고 한다. 검색된 *k*개의 객체가 *k*-최근접 객체임을 보장하는지 확인하기 위해 추가적으로 *k-dist*보다 질의 점에 가까운 모든 셀들의 탐색을 순서대로 진행한다. 만약 추가적으로 발견된 객체가 *k-dist*보다 질의 점에 더 가까우면, 결과 셋과 *k-dist*를 갱신한다. 남은 셀들이 *k-dist*보다 질의 점에 가까운 것이 없으면 처리를 중단한다.

그림 5는 *CLR*을 이용하여 3-최근접 질의를 처리하는 과정을 보여준다. 그림 3.12(a)는 3개의 객체를 찾기 위해서 *CLR*을 순서대로 적용하여 셀을 탐색하는 것을 보여준다. 그림 3.12(b)는 3-최근접 객체임을 보장하는 3개의 객체를 찾는 과정을 보여준다. 어두운 부분은 질의에 의해서 탐색된 셀들을 나타낸다.

Collection @ chungbuk

(a) CLR을 이용한 3개의 객체 탐색 (b) 3-NN을 보장하는 객체 탐색

그림 3.12 거리 관계 패턴을 이용한 3-최근접 질의 처리

하지만 *CLR*에는 오차가 있기 때문에 추가적인 셀 탐색이 필요하다. *CLR*상의 다음 순서에 탐색할 셀들이 이전 순서에 탐색한 셀들보다 질의 점에 근접할 수 있다. 이것은 *CLR*이 담당하는 영역에 존재하는 모든 점에 대해서 근접한 셀의 순서를 만족 시키지 않기 때문이다. 오차를 극복하기 위해 *search-dist*를 이용하여 셀을 탐색한다. *EQ(q)*는 질의 *q*에 대해서 *k-dist*의 오차 값을 나타낼 때, *search-dist*는 식 3.6과 같다.

$$search_dist(q) = q.\ k\text{-}dist + EQ(q) \qquad (3.6)$$

그림 3.13은 *E*와 *EQ(q)*의 차이 그리고 *search-dist*와 *k-dist*의 차이를 보여준다. *k*-최근접 객체를 보장하기 위해 *CLR*에서 제공하는 순서로 셀 탐색 시, *search-dist*와 탐색하는 셀과의 거리를 비교하면 거리 관계 패턴의 오차가 고려된다. 하지만 실제 셀을 탐색할 때는 질의 점으로부터의 거리가 *k-dist*보다 작은 셀만을 탐색한다.

Collection @ chungbuk

그림 3.13 *search-dist*를 고려한 검색 영역

그림 3.14는 *k*-최근접 질의를 연산하는 알고리즘이다. 일단 주어진 질의 *q*가 내려지면 질의 *q*에 해당되는 *CLR clr*을 구하고(줄 03), 질의 *q*의 좌표를 포함하는 셀 C_{cur}을 구한다(줄 04). $q.p_{clr}$을 1씩 증가시켜 가면서 $clr[q.p_{clr}]$을 셀 C_{cur}의 좌표에 적용하여 근접한 셀을 순서대로 탐색하고, *k*개의 객체를 찾아 *result_set*에 삽입한다(줄 05~07). *k*개의 객체를 찾은 후 *q.k-dist*를 계산한다(줄 08). *k*개의 객체가 *k*-최근접 객체임을 보장하기 위해 $q.p_{clr}$을 1씩 증가시켜 가면서 $clr[q.p_{clr}]$에 따라 $search_dist(q) > clr[q.p_{clr}].dist$ 조건을 만족할 때까지 근접한 셀들을 추가적으로 탐색한다(줄 09). 만약 *q.k-dist*보다 근접한 객체가 발견되면 *q.result_set* 그리고 q.*k-dist*를 모두 갱신한다(줄 16~20). 줄 10은 연속 질의 처리 시에 점진 적인 처리를 위해 질의와 겹치는 모든 셀에 *q.qid*를 삽입하는 것을 나타낸다.

Collection @ chungbuk

거리 관계 패턴을 이용한 *k*-최근접 질의 처리 기법을 이용하여 연속 질의를 효율적으로 처리하기 위해 객체 색인과 더불어 질의 색인이 필요하다. 그림 3.15는 그리드 색인에서 유지하는 자료구조를 나타낸다. 각 셀마다 *oid_list*와 *qid_list*를 유지한다. oid_list는 그 셀의 영역에 있는 객체들의 식별자 *oid*를 유지하고, *qid_list*는 그 셀의 영역과 질의 점으로부터 *k-dist*반경의 원에 겹치는 질의들의 식별자 *qid*를 유지한다. *querytable*에는 질의 식별자 *qid,* 검색하고자 하는 객체 개수 *k*, *k*번째 객체까지의 거리 *k-dist*, 질의 결과 *result_set* 그리고 *CLR*상의 마지막으로 방문한 위치 p_{clr}를 유지한다.

객체는 자신의 위치를 주기적으로 서버로 보고하고, 서버는 매 주기마다 질의의 결과를 갱신한다. 질의 마다 *k-dist* 안으로 들어온 객체의 리스트와 개수 그리고 밖으로 나간 객체의 수를 유지한다. 만약 들어온 객체가 동일하거나 많으면 들어온 객체의 리스트를 결과에 적용하여 *result_set*과 *k-dist*를 갱신한다. 만약 나간 객체들이 많거나 질의가 이동하면, *k*-최근접 질의를 재 수행한다.

Collection @ chungbuk

```
Algorithm kNN_Computation(Query q)
// Input =   q: 질의
// Output = 질의 결과
  clr = HCLR(q)
  Cq = QC(q)
  While (k > |q.result_set|)
    Ccur  = AC(Cq,clr[q.pclr])
    Foreach object o ∈ OL(Ccur)
      Insert o into q.result_set;
    q.pclr++
  Update q.k-dist
  While (search_dist(q) > clr[q.pclr].dist)
    Insert q into QL(Ccur)    /*forcontinuousqueries*/
    If (q.k-dist < dist(q,Ccur))
      q.pclr++
      continue
    Ccur  = AC(Cq,clr[q.pclr])
    Foreach o, ∀o ∈ OL(Ccur)
      If (q.k-dist > dist(o,q))
        Remove o^(k-th) from q.result_set
        Insert o into q.result_set
        Update q.k-dist
    q.pclr++
  Return q.result_set remove O from OL(q), ∀q ∈ (gold.FL ∪ QL(cold));
```

그림 3.14 k-최근접 질의 처리 알고리즘

그림 3.15 연속 k-최근접 질의 처리를 위한 자료구조

Collection @ chungbuk

3.3.3 객체 이동 시 연속 *k*-최근접 질의 처리

경우 1. 주어진 질의 q가 있고, 시간 t_0에 객체 o가 $dist(o,q) \geq q.k\text{-}dist$ 을 만족하고, 시간 t_1에 $dist(o,q) < q.k\text{-}dist$ 를 만족한다.

경우 1은 객체 o가 이동하여 질의 q의 $k\text{-}dist$ 반경으로 들어간 경우를 나타낸다. 질의 처리를 위해서 질의 $q_i.result_set$에서 k번째 객체를 삭제하고, 객체 o를 $q.result_set$에 삽입한다. 그리고 $q.k\text{-}dist$를 갱신한다. 마지막으로 CLR을 $q.p_{clr}$로부터 역방향으로 순회하여 $q.k\text{-}dist \leq dist(q, c)$를 만족하는 셀 c의 질의 리스트 $QL(c)$에서 $q.qid$를 제거한다. CLR을 역방향으로 순회 시, 결과 값을 보장하기 위해 $reverse_search_dist$를 이용한다. $reverse_search_dist$는 식 3.7과 같다.

$$reverse_search_dist(q) = q.k\text{-}dist - EQ(q) \tag{3.7}$$

그림 3.16는 주어진 질의 q의 $k\text{-}dist$안으로 객체 o가 이동한 경우에 점진 적인 연속질의 처리 알고리즘을 나타낸다. 질의 q의 결과 집합에서 k번째 객체를 삭제하고, 객체 o를 그 결과 집합에 삽입한다(줄01~02). 그리고 $q.k\text{-}dist$를 갱신한다(줄03). 질의 q에 해당되는 CLR clr을 구하고(줄04), 질의 q의 좌표를 포함하는 셀을 구한다(줄05). 질의 q의 k번째 객체가 삭제되어 $q.k\text{-}dist$가 줄어들었기 때문에 질의의 영역을 줄이는 과정이 필요하다. 이 과정은 $q.p_{clr}$을 1씩 감소시켜 가면서 $clr[q.p_{clr}]$에 따라 $reverse_search_dist(q) < clr[q.p_{clr}].maxdist$ 조건을 만족할 때 까지 셀을 탐색하면서 질의의 영향을 더

Collection @ chungbuk

이상 받지 않는 셀 c_i의 질의 리스트 $QL(c_i)$에서 질의 식별자 $q.qid$를 삭제한다(줄 06~11). 다시 말해서 탐색된 셀 c_i에 대해서 $q.k\text{-}dist \le dist(q, c_i)$를 만족하면 $QL(c_i)$에서 그 셀에서 $q.qid$를 삭제한다.

그림 3.17은 연속 3-최근접 질의 q에 대한 경우 1의 예를 나타낸다. 그림 3.17(a)는 시간 t_0에서 객체의 상황을 나타내고 질의 q의 k번째 객체는 o_1이기 때문에 q의 $k\text{-}dist$는 $dist(q, o_1)$이다. 그림 3.17(b)는 시간 t_1에 객체 o_4가 질의 q의 $k\text{-}dist$ 반경 안으로 이동한 것을 나타낸다. 이러한 경우 시간 t_0에서의 질의 q의 $result_set$인 o_1, o_2, o_3에서 k번째 객체 o_1을 삭제하고, 객체 o_4를 삽입한다. 그리고 질의 q의 $k\text{-}dist$를 $dist(q, o_2)$로 갱신한다. 마지막으로 시간 t_0에서는 질의 q의 $k\text{-}dist$ 반경과 겹쳤지만, 시간 t_1에서는 더 이상 질의 q의 $q.k\text{-}dist$반경과 겹치지 않는 셀 c의 질의 리스트 $QL(c)$에서 $q.qid$를 삭제한다. 그 셀들은 그림 3.17(b)에서 어둡게 표시되었다.

```
    Algorithm CkNN_object_move_into(q, o)
    // Input =   q: 질의, o: 객체
    // Output = 질의 결과
01    Remove k-th object from q.result_set
02    Insert o into q.result_set
03    Update q.k-dist
04    clr = HCLR(q)
05    Cq = QC(q)
06    While (reverse_search_dist(q) < clr[q.pclr].maxdist)
07      Ccur  = AC(Cq,clr[q.pclr])
08      q.pclr--
09      If (q.k-dist ≤ dist(q, Ccur))
10        Remove q from Ccur.qid_list
11    Return q.result_set
```

그림 3.16 질의 q의 $k\text{-}dist$안으로 객체 o가 이동한 경우의 알고리즘

Collection @ chungbuk

(a) 객체 o4 이동 전 (t0)　　(b) 객체 o4 이동 후 (t1)

그림 3.17 질의 q의 영역으로 객체 o_4의 이동

경우 2. 주어진 질의 q가 있고, 시간 t_0에 객체 o가 $dist(o,q) \leq q.k\text{-}dist$ 을 만족하고, 시간 t_1에 $dist(o,q) > q.k\text{-}dist$ 를 만족한다.

경우 2는 객체 o가 이동하여 질의 q의 $k\text{-}dist$반경 밖으로 벗어난 경우를 나타낸다. 질의 처리를 위해서 CLR을 $q.p_{clr}$로부터 역방향으로 순회하여 질의 결과에 영향을 미칠 수 있는 모든 셀들을 탐색한다. 다시 말해서, $k\text{-}dist$반경 안에 있는 객체는 질의가 이미 알고 있기 때문에 그 반경에 완전히 포함되는 셀은 탐색할 필요가 없기 때문에 $k\text{-}dist$ 반경에 부분적으로 겹치는 모든 셀들을 탐색한다. 그리고 다시 CLR을 $q.p_{clr}$로부터 정방향으로 순회하여 질의를 처리한다. 이때 $dist(q,c) < q.k\text{-}dist$를 만족하는 셀 c의 질의 리스트 $QL(c)$에 $q.qid$를 삽입한다.

Collection @ chungbuk

그림 3.18은 주어진 질의 q의 k-$dist$ 밖으로 객체 o가 벗어난 경우에 점진적인 연속 질의 처리 알고리즘을 나타낸다. 우선 $q.result_set$에서 객체 o를 삭제한다(줄 01). 질의 q에 해당되는 CLR clr을 구하고(줄02), 질의 q의 좌표를 포함하는 셀을 구한다(줄03). 그리고 그림 3.16의 알고리즘에서와 비슷한 방식으로 CLR을 역으로 순회한다(줄 07). 하지만 질의 q의 k-$dist$ 반경에 완전히 포함되는 셀의 모든 객체들은 이미 $q.result_set$에 포함되어 있고, 질의 결과에 영향을 미치지 않기 때문에 연산에서 제외시켜야 하기 위해서 $k\text{-}dist_{old} > maxdist(q,c)$를 만족하지 셀 c를 연산에서 제외시킨다(줄 08~10). $k\text{-}dist_{old}$는 질의 처리가 수행되기 전의 질의의 k-$dist$ 값을 나타낸다(줄 06). CLR의 역방향 순회를 마치면, 정방향 순회를 통해 질의 처리를 계속한다. 정방향 순회를 위해서 그림 3.14의 $kNN_Computation()$ 알고리즘을 이용한다(줄 18).

그림 3.19는 연속 3-최근접 질의 q에 대한 경우 2의 예를 나타낸다. 그림 3.19(a)는 시간 t_0에서 객체의 상황을 나타낸다. 질의 q의 k번째 객체는 o_1이기 때문에 q의 k-$dist$는 $dist(q, o_1)$이다. 그림 3.19(b)는 시간 t_1에 객체 o_3가 질의 q의 k-$dist$ 반경 밖으로 이동한 것을 나타낸다. 점선으로 표현된 원은 시간 t_0의 k-$dist$ 반경의 원이고, 실선으로 표현된 원은 시간 t_1의 k-$dist$ 반경으로 그린 원이다. 시간 t_0의 k-$dist$를 $k\text{-}dist_{t0}$로 나타내고, 시간 t_1의 k-$dist$를 $k\text{-}dist_{t1}$로 나타낼 때, $k\text{-}dist_{old}$와 겹치는 셀들은 CLR의 역방향 순회로부터 탐색되었고, $k\text{-}dist_{t0}$와 $k\text{-}dist_{t1}$사이에 있거나 $k\text{-}dist_{t1}$에 겹치는 셀들은 CLR의 정방향 순회로부터 탐색되었다. 그림 3.19(b)에서 질의 처리에 참여한 셀들은 어둡게 표시되었다.

Collection @ chungbuk

```
Algorithm CkNN_object_move_out(q, o)
// Input =   q: 질의, o: 객체
// Output = 질의 결과
  Remove o from q.result_set
  clr = HCLR(q)
  C_q = QC(q)
  p_temp = q.p_clr
  C_cur = AC(q, clr [p_temp])
  k-dist_old = q.k-dist
  While (reverse_search_dist(q) < clr[p_clr].maxdist)
    If (k-dist_old > maxdist(q,C_cur))
      p_temp--
      coutinue
    C_cur = AC(q, clr [p_temp])
    For each object o, ∀o ∈ OL(C_cur)
      If (q.k-dist > dist(o,q))
        Remove k-th object from q.result_set
        Insert o into q.result_set
        Update q.k-dist
    p_temp--
  kNN_Computation(q)
  Return q.result_set
```

그림 3.18 질의 q의 k-$dist$ 밖으로 객체 o가 이동한 경우의 알고리즘

Collection @ chungbuk

(a) 객체 이동 전(t0) (b) 객체 이동 후(t1)

그림 3.19 질의 q의 영역 밖으로 객체 o_3의 이동

Collection @ chungbuk

3.3.4 질의 이동 시 연속 *k*-최근접 질의 처리

정의 8. 주어진 질의 q에 대해서 이동 전의 질의를 q_{old},이동 후의 질의를 q_{new},그리고 질의가 이동한 거리 즉 $dist(q_{old},q_{new})$를 m이라고 할 때, 식 3.8를 만족하면 $q_{old}.result_set$과 $q_{new}.result_set$은 동일하다는 것이 보장된다.

$$m \geq q_{old}.k\text{-}dist \ - \ q_{new}.k\text{-}dist \tag{3.8}$$

그림 3.20은 정의 8에 대한 예제를 나타낸다. 질의 q가 q_{old}에서 q_{new}로 이동하였다. q_{old}와 q_{new}의 거리는 $q_{old}.k\text{-}dist$와 $q_{new}.k\text{-}dist$의 차와 같다. 이러한 경우 q_{old}의 $k\text{-}dist$ 반경에 q_{new}의 $k\text{-}dist$반경이 완전히 포함되므로 $q_{new}.result_set$에 포함되는 k개의 객체는 모두 $q_{old}.result_set$에 포함되므로 q_{old}와 q_{new}는 동일한 결과값을 가지게 된다. 그림 3.20에서 질의 q는 식 8을 만족하므로 q_{old}와 q_{new}의 k-최근접 객체는 동일하게 o_1, o_2그리고 o_3을 갖는 것을 보여준다.

그림 3.20 식 3.8을 만족하는 질의 이동 예제

Collection @ chungbuk

경우 1. 주어진 질의 q가 이동하고, 시간 t_0에 질의 q와 시간 t_1에 질의 q의 위치가 한 *VDC*영역에 존재한다.

경우 1은 질의 q가 한 *VDC*영역 내에서 이동한 경우를 나타낸다. 이때 질의의 p_{clr}값은 여전히 유효하기 때문에 *CLR*을 이용한 점진 적인 질의 처리가 가능하다. 만약 식 3.8을 만족하면 이 경우 질의 처리 알고리즘은 경우 1의 *CkNN_object_move_into*()와 유사하게 처리되고, 반면에 그 조건을 만족하지 않은 경우 질의 처리 알고리즘은 경우 2의 *CkNN_object_move_out*()와 유사하게 처리된다.

그림 3.21은 질의 q가 한 *VDC*영역 내에서 이동한 경우의 알고리즘을 나타낸다. $m \geq (q_{old}.k\text{-}dist - q_{new}.k\text{-}dist)$을 만족하면 *CkNN-object_move_into*(q_{new}, *null*)을 수행한다(줄 02~03). 그렇지 않으면 q_{new}의 *result_set*에서 $dist(o, q_{new}) < q_{old}.k\text{-}dist$를 만족하는 모든 객체를 삭제하고, *CkNN_object_move_out*(q_{new}, *null*)을 수행한다(줄 05~08).

경우 2. 주어진 질의 q가 이동하고, 시간 t_0에 질의 q와 시간 t_1에 질의 q의 위치가 다른 *VDC*영역에 존재한다.

경우 2는 질의 q가 한 *VDC*영역 밖으로 이동한 경우를 나타낸다. 질의를 초기화 하고, 질의를 재 수행한다. 알고리즘은 그림 3.22에서 보여준다. 일단 질의 q를 초기화하고(줄 01), *kNN_Computation*(q)를 수행한다(줄 02).

Collection @ chungbuk

```
Algorithm CkNN_query_move_within_a_VDC(q_old, q_new, m)
// Input = q_old, q_new:이동 전과 후의 질의, m: 질의 이동 거리
// Output = 질의 결과
  Updata q_new.k-dist
  If (m ≥ (q_old.k-dist-q_new.k-dist))
    CkNN-object_move_into(q_new, null)
  Else
    Foreach object o, ∀o ∈ q_new.result_set
      If (dist(o,q_new) < q_old.k-dist)
        Remove o from q_new.result_set
    CkNN_object_move_out(q_new, null)
  Return q_new.result_set
```

그림 3.21 질의 q가 한 VDC영역 내에서 이동한 경우의 알고리즘

```
Algorithm CkNN_query_move_out_of_a_VDC(q)
// Input =   q: 질의
// Output =   result_set : k-최근접 결과 집합
  Initialize q
  kNN_Computation(q)
  Return q.result_set
```

그림 3.22 질의 q가 한 VDC영역 밖으로 이동한 경우의 알고리즘

Collection @ chungbuk

3.3.5 다수의 객체 이동과 질의 이동에 따른 질의 처리기법

이동객체 환경에서는 많은 객체가 동시에 이동하는 경우를 고려해야만 한다. 그렇기 때문에 한 객체의 이동마다 연속 질의를 처리하는 것은 매우 비효율적이다. 본 절에서는 동시에 다수의 객체가 이동했을 경우의 질의 처리 기법을 제안한다. 동시에 이동한 객체에 대해서는 바로 질의처리를 수행하지 않고 버퍼에 저장하여 한 번에 처리한다. 이때 질의 q는 *q.k-dist* 안으로 들어온 객체들의 리스트 *q.ilist*와 그 객체들의 수 *q.inum*, 그리고 밖으로 벗어난 객체들의 수 *q.onum*을 기록한다. 그래서 질의 처리 시, *q.inum*≥ *q.onum*이면, 질의 q의 영역은 작아지며, 그렇지 않으면 질의 *q*의 영역은 커진다.

그림 3.23은 다수 객체 이동에 따른 질의 처리 기법을 나타낸다. 질의 *q*의 *k-dist* 안으로 들어온 객체들은 *q.result_set*에 삽입하고, *q.result_set*에서 *k*개의 최근접 객체를 제외한 객체들을 모두 삭제한다(줄 01~04). 만약 들어온 객체의 수가 더 많거나 같으면 *CkNN_object_move_into(q, null)*를 수행하고, 그렇지 않으면 *CkNN_object_move_out(q, null)*를 수행한다(줄 05~08).

```
    Algorithm CkNN_multi_objects_move(q)
    // Input =   q: 질의
    // Output =   result_set : k-최근접 결과 집합
01      Insert q.ilist into q.result_set
02      Foreach o, ∀o ∈ q.result_set
03        If (dist(o, q) < q.k-dist)
04          Remove o from q.result_set
05      If (q.inum ≥ q.onum)
06        CkNN-object_move_into(q, null)
07      Else
08        CkNN_object_move_out(q, null)
```

그림 3.23 다수 객체가 이동한 경우의 알고리즘

Collection @ chungbuk

다수 객체와 함께 질의 또한 동시에 이동을 한다. 이러한 경우 객체 이동에 따른 질의 갱신을 수행하고 다시 질의 이동에 따른 갱신을 수행하게 되면 비효율적이다. 본 절에서는 객체의 이동과 질의의 이동을 모두 고려한 알고리즘을 소개한다.

그림 3.24은 다수 객체 이동과 질의 이동을 고려한 알고리즘을 나타낸다. 주어진 질의 q에 대해서 이동전의 질의를 q_{old}그리고 이동 후의 질의를 q_{new}로 표현한다. 우선 질의 q의 k-$dist$ 안으로 들어온 객체들을 $q.result_set$에 삽입한다(줄 01). 만약 질의 q가 한 VDC 내에서 이동했다면, $q_{new}.k$-$dist$를 갱신하고, $q_{new}.result_set$에서 k개의 최근접 객체를 제외한 객체들을 삭제한다(줄 03~06). 다음으로 $CkNN_query_move_within_a_VDC(q_{old}, q_{new}, dist(q_{old}, q_{new}))$를 수행시킨다(줄 07). 만약 질의 q가 한 VDC 밖으로 이동했다면 아무런 사전 연산 없이 $CkNN_query_move_out_of_a_VDC(q_{new}, dist(q_{old}, q_{new}))$를 수행시킨다(줄 09).

결론적으로, 질의가 이동하지 않은 경우에는 $CkNN_multi_objects_move(q_{old}, q_{new})$를 수행시키고, 질의가 이동한 경우에는 $CkNN_multi_objects_move_with_queries(q_{old}, q_{new})$를 수행시켜서 효율적인 연속 k-최근접 질의를 처리한다.

Collection @ chungbuk

```
Algorithm CkNN_multi_objects_move_with_queries(q_old, q_new)
// Input = q_old, q_new:이동 전과 후의 질의, m: 질의 이동 거리
// Output =   result_set : k-최근접 결과 집합
  Insert ∃o ∈ q_new.ilist into q_new.result_set
  If (q_old and q_new are within a VDC)
    Update q_new.k-dist
    Foreach o, ∀o ∈ q_new.result_set
      If (dist(o, q_new) < q_new.k-dist)
        Remove o from q_new.result_set;
    CkNN_query_move_within_a_VDC(q_old, q_new, dist(q_old,q_new))
  Else
    CkNN_query_move_out_of_a_VDC(q_new, dist(q_old, q_new))
```

그림 3.24 다수 객체 이동과 질의 이동을 고려한 알고리즘

Collection @ chungbuk

IV. 분석

이 장에서는 대표적인 k-최근접 질의 처리 기법인 CPM과 제안하는 질의 처리기법인 PB-kNN의 저장 비용과 연산비용을 비교 분석한 결과를 설명한다.

4.1 PB-kNN의 저장 비용

PB-kNN이 요구하는 저장 공간 $S_{pb\text{-}knn}$은 식 4.1에서 나타낸다.

$$S_{pb-knn} = S_{grid} + S_{object_table} + S_{query_table} + S_{drp} \tag{4.1}$$

S_{grid}는 그리드 색인, S_{object_table}는 객체 테이블 그리고 S_{query_table}는 질의 테이블이 요구하는 저장 공간을 나타낸다. S_{drp}는 거리 관계 패턴이 요구하는 저장 공간의 크기를 나타낸다. S_{grid}는 각 셀이 요구하는 저장 공간 S_{cell}의 크기에 의해 결정된다. 셀의 개수가 n일 때, 식 4.2과 같다.

$$S_{grid} = \sum_{i=0}^{n} S_{cell}(i) \tag{4.2}$$

Collection @ chungbuk

S_{cell}은 각 셀이 가지는 포인터의 크기 S_{ptr}, 객체 리스트의 크기 S_{object_list} 그리고 질의 리스트의 크기 S_{query_list}를 포함한다. 셀 번호가 c인 셀의 저장 공간은 식 4.3과 같다.

$$S_{cell}(c) = S_{ptr} + S_{object_list}(c) + S_{query_list}(c) \tag{4.3}$$

객체 테이블의 크기 S_{object_table}는 메모리에서 유지하는 각 객체 정보의 크기 S_{object}에 의해 결정된다. 전체 객체 집합을 O라고 할 때, 식 4.4와 같다.

$$S_{object_table} = \sum_{\forall o \in O} S_{object}(o) \tag{4.4}$$

그리고 한 객체 o가 요구하는 저장 공간 $S_{object}(o)$은 식 4.5과 같다

$$S_{object}(o) = S_{oid} + S_{coordinate} \tag{4.5}$$

PB-kNN에서는 객체 정보로서 객체 식별자와 위치 좌표만을 사용한다. S_{oid}는 객체 식별자의 크기를 나타내고, $S_{coordinate}$는 객체 위치 좌표 정보의 크기를 나타낸다.

Collection @ chungbuk

질의 테이블의 크기 $S_{query_table_pb}$는 각 질의 정보의 크기 S_{query_pb}에 의해 결정된다. 전체 질의 집합을 Q라고 할 때, 식 4.6와 같다.

$$S_{query_table_pb} = \sum_{\forall q \in Q} S_{query_pb}(q) \tag{4.6}$$

그리고 한 질의 q가 요구하는 저장 공간 $S_{query_pb}(q)$은 식 4.7와 같다.

$$S_{query_pb}(q) = S_{qid} + S_{coordinate} + S_{result} + S_{pos_value} \tag{4.7}$$

PB-kNN에서는 질의 정보로서 질의 식별자, 질의 위치 좌표, 결과 집합, 그리고 CLR 상의 위치를 나타내는 pos 값을 사용한다. S_{qid}는 질의 식별자의 크기, $S_{cooridinate}$는 질의 위치 좌표 정보의 크기, S_{result}는 질의 결과 집합의 크기, 그리고 S_{pos_value}는 pos값의 크기를 나타낸다. S_{result}는 질의의 k값에 의해 결정되고, 그 크기는 식 4.8에서 나타낸다.

$$S_{result} = S_{oid} \times k \tag{4.8}$$

Collection @ chungbuk

마지막으로 거리 관계 패턴의 크기 S_{drp}는 식 4.9과 같다.

$$S_{drp} = S_{coordinate} \times N_{clr} \times N_{coordinates} \tag{4.9}$$

N_{clr}은 만들어진 CLR의 개수를 나타내고, $N_{coordinate}$는 거리 관계 패턴을 위한 상대 좌표의 개수를 나타낸다. 만약 $N_{coordinate}$가 100이면 각 CLR은 100개의 가장 근접한 셀의 상대 좌표만을 가진다.

Collection @ chungbuk

4.2 CPM과 저장 비용

CPM이 요구하는 저장 공간 S_{cpm}은 식 4.10에서 나타낸다. .

$$S_{cpm} = S_{grid} + S_{object_table} + S_{query_table_cpm} \tag{4.10}$$

PB-kNN과 CPM은 동일한 그리드 색인과 객체 테이블을 가진다. 그래서 S_{grid}과 S_{object_table}은 두 알고리즘에서 같은 값을 갖는다. 반대로 말하면 PB-kNN과 다른 질의 테이블을 가진다. CPM의 질의 테이블 크기 $S_{query_table_cpm}$는 각 질의 정보의 크기 S_{query_cpm}에 의해 결정된다. 그 수식은 식 4.11에서 나타낸다.

$$S_{query_table_cpm} = \sum_{\forall q \in Q} S_{query_cpm}(q) \tag{4.11}$$

그리고 한 질의 q가 요구하는 저장 공간 $S_{query}(q)$은 식 4.12에서 나타낸다.

$$S_{query_cpm}(q) = S_{qid} + S_{coordinate} + S_{result} + S_{visit_list}(q) + S_{priority_queue}(q) \tag{4.12}$$

$S_{priority_queue}(q)$는 질의 식별자가 qid인 질의가 질의 처리 시, 사용하는 우선순위 큐의 크기를 나타낸다. $S_{visit_list}(q)$는 질의 식별자가 qid인 질의가 순서대로 방문한 셀의 번호들의 리스트의 크기를 나타낸다.

Collection @ chungbuk

4.3 PB-*k*NN과 CPM의 저장 비용 비교

CPM는 *k*-최근접 질의 처리를 위해 우선순위 큐를 이용하고, 연속 질의 처리를 위해 셀 방문리스트와 그 우선순위 큐를 각 질의 마다 유지한다. 이 자료구조는 질의 처리를 위해 방문하는 셀의 개수가 많아질수록 더 많은 저장 공간의 낭비를 초래한다.

PB-kNN은 *k*-최근접 질의 처리를 위해 CPM과 다르게 저장 비용의 낭비를 초래하는 셀 방문 리스트와 우선순위 큐를 사용하지 않는다. 대신에 본 논문에서 제안된 거리 관계 패턴을 이용하여 질의를 처리한다. PB-*k*NN이 CPM에 비해 줄일 수 있는 저장 공간의 크기 S_{save}는 식 4.13에서 나타낸다.

$$S_{save} = \sum_{\forall q \in Q} (S_{visit_list}(q) + S_{priority_queue}(q)) - S_{drp} \tag{4.13}$$

질의의 개수가 늘어날수록 또는 객체의 개수가 줄어들어서 질의가 방문하는 셀의 개수가 늘어날수록 PB-*k*NN이 CPM에 비해 절약하는 저장 공간의 크기는 더욱 커진다. 이러한 특징은 5장 성능평가에서 보여준다.

Collection @ chungbuk

4.4 PB-kNN의 연산 비용

PB-kNN이 k-최근접 질의 처리 시 요구하는 연산 비용 $C_{pb\text{-}knn}$은 식 4.14에서 나타낸다.

$$C_{pb-knn} = C_{dist_cells} + C_{dist_objects} + C_{adapt_drp} \tag{4.14}$$

$C_{pb\text{-}knn}$은 질의와 셀과의 거리를 연산하는 비용 C_{dist_cells}와 k-최근접 객체를 확인하기 위해 질의와 객체와의 거리를 연산하는 비용 $C_{dist_objects}$ 그리고 근접한 셀을 찾기 위해 거리 관계 패턴의 상대 좌표를 적용하는 비용 C_{adapt_drp}으로 구성된다.

C_{dist_cells}는 한 셀과 질의와의 거리 측정비용을 $C_{dist_a_cell}$라고 하고, 질의 처리에 의해 접근된 셀의 개수를 N_{visit_cells}라고 할 때, 식 4.15과 같다.

$$C_{dist_cells} = C_{dist_a_cell} \times N_{visit_cells} \tag{4.15}$$

$C_{dist_objects}$는 한 객체와 질의와의 거리 측정 비용을 $C_{dist_a_object}$로 하고, 한 셀의 포함되는 평균 객체의 개수를 $N_{objects_in_cell}$로 표기할 때, 식 4.16와 같다.

$$C_{dist_objects} = C_{dist_a_object} \times N_{objects_in_cell} \times N_{visit_cells} \tag{4.16}$$

Collection @ chungbuk

C_{adapt_drp}는 질의를 포함하는 셀의 좌표에 CLR상의 상대 좌표를 적용하는 비용을 $C_{adapt_coordinate}$라고 할 때, 식 4.17에서 표현한다.

$$C_{adapt_drp} = C_{adapt_coordinate} \times N_{visit_cells} \tag{4.17}$$

결국 $C_{pb\text{-}knn}$은 식 4.18과 같이 표현한다.

$$C_{pb-knn} = N_{visit_cells}(C_{dist_a_cells} + C_{dist_a_object} \times N_{objects_in_cell} + C_{adapt_coordinate}) \tag{4.18}$$

k개의 최근접 객체를 구하기 위해 탐색하는 셀의 개수 N_{visit_cells}는 식 4.19과 같다.

$$N_{visit_cells} = \left\lceil \frac{k}{N_{objects_in_cell}} \times \alpha \right\rceil \qquad (\text{단}, \alpha \geq 1) \tag{4.19}$$

N_{visit_cells}는 k개의 객체를 포함하는 면적과 비례하기 때문에 그 면적에 계수 a를 곱하여 구한다.

Collection @ chungbuk

객체의 수를 $|O|$로 나타내고, 그리드의 셀 개수를 $|C|$로 나타낸다. $N_{objects_in_cell}$은 식 4.20과 같다.

$$N_{objects_in_cell} = \frac{|O|}{|C|} \tag{4.20}$$

그래서 N_{visit_cells}는 식 4.21와 같다.

$$N_{visit_cells} = \left\lceil \frac{k \cdot |C|}{|O|} \times \alpha \right\rceil \tag{4.21}$$

식 4.19, 식 4.20 그리고 식 4.21를 $C_{pb\text{-}knn}$에 대한 식 4.18에 대입하면 식 4.22으로 표현된다.

$$C_{pb-knn} = \left\lceil \frac{k \cdot |C|}{|O|} \times \alpha \right\rceil (C_{dist_a_cells} + C_{dist_a_object} \times \frac{|O|}{|C|} + C_{adapt_coordinate}) \tag{4.22}$$

위의 식은 k값과 연산 비용은 비례한다는 것을 나타낸다. 하지만 $|O|$와 $|C|$에 의해서 결정되는 $N_{objects_in_cell}$이 너무 크거나 작으면 연산비용의 증가를 초래한다. 최적의 성능을 위해서는 $|O|$와 $|C|$에 대해서 적절한 값의 결정이 필요하다.

Collection @ chungbuk

4.5 CPM의 연산 비용

CPM이 k-최근접 질의 처리 시 요구하는 연산 비용 C_{cpm}은 식 4.23에서 나타낸다.

$$C_{cpm} = C_{dist_cells} + C_{dist_objects} + C_{priority_queue} \tag{4.23}$$

$C_{pb\text{-}knn}$와 다른 점은 C_{adapt_drp}의 부재와 $C_{priority_queue}$의 추가이다. $C_{priority_queue}$는 CPM이 질의 처리 시에 우선순위 큐를 유지하는데 소모되는 비용이다. 식 4.24에서 그 비용을 나타낸다.

$$C_{priority_queue} = (C_{enqueue} + C_{dequeue}) \times N_{visit_cells} \tag{4.24}$$

$C_{enqueue}$와 $C_{dequeue}$는 우선순위 큐에 셀을 넣고 꺼내는 비용을 각각 나타낸다. CPM에서는 k-최근접 질의를 처리하기 위해서 탐색한 셀의 개수만큼 $C_{enqueue}$와 $C_{dequeue}$ 비용을 소모하기 때문에 그 비용을 주된 비용으로 고려한다.

Collection @ chungbuk

4.6 PB-*k*NN과 CPM의 연산 비용 비교

PB-*k*NN과 CPM의 차이를 명확히 하기 위해서 PB-kNN이 CPM에 비해서 절약하는 연산 비용 C_{save}을 식 4.25에서 표현한다.

$$C_{save} = (C_{enqueue} + C_{dequeue} - C_{adapt_coordinate}) \times N_{visit_cells} \qquad (4.25)$$

$C_{enqueue}$와 $C_{dequeue}$의 평균 비용은 $O(log_2 n)$으로 가정한다. 이때 n는 정렬을 수행할 때 큐에 있는 객체의 개수이다. 반면에 $C_{adapt_coordinate}$는 단순히 좌표의 각 차원에 덧셈 연산을 수행한 것이므로 단지 두 번의 덧셈 연산 비용이 필요하다. 그러므로 탐색하는 셀의 개수가 증가할수록 그 비용차이는 커지게 된다.

Collection @ chungbuk

V. 성능 평가

5.1 연속 영역 질의 처리 기법 성능 비교

5.1.1 평가 환경

제안하는 연속 영역 질의 처리 기법의 우수성을 입증하기 위해 펜티엄4 CPU 3GHz, RAM 1Gbyte, 윈도우즈 XP 운영체제에서 C언어로 구현하여 성능 평가를 수행하였다. 성능 평가를 위해 사용된 데이터는 비교 대상인 CES-Index[7]에서 사용된 데이터와 동일한 조건으로 생성하였다. 성능 평가를 위해 전체 그리드 크기를 512×512로 하고 8,000개의 질의와 64,000개의 객체를 생성한다. 성능 평가에 많은 영향을 미치는 질의의 크기와 이동 객체의 속도는 다양한 값으로 변화하면서 성능 평가를 수행한다. 표 5.1은 성능 평가를 위해 사용된 인자들을 나타낸 것이다. 표 5.1에 나타난 인자의 크기는 최소 셀 크기에 비례한다. 본 논문에서 연속 영역 질의 처리를 위해 제안하는 색인 기법은 편의상 BID로 나타내었다.

표 5.1 성능 평가 인자

인자	내용
질의의 크기	1 - 80
객체 이동 속도	0 - 20
그룹의 크기	4x4 - 64x64

Collection @ chungbuk

5.1.2 그룹 크기와 비트 식별자 크기에 따른 성능 변화 분석

BID 기법에서 그룹 크기는 성능을 결정하는 아주 중요한 요소이다. 그룹의 크기가 증가할수록 부분적으로 겹치는 질의의 수가 증가하므로 그룹 내의 셀들이 유지하는 비트 열의 크기가 질의의 수만큼 증가된다. 비트 열의 크기가 너무 크면 객체의 이동에 따른 갱신 처리 시 비트 열 비교 연산의 증가와 저장 공간의 낭비를 초래할 수 있다. 또한 그룹의 크기가 감소할수록 부분적으로 겹치는 질의의 수는 감소되어 비트 열의 크기는 작아지지만, 그룹들의 중복적인 질의 식별자 저장 횟수가 증가되고 그룹을 벗어나는 이동 객체의 갱신이 많아지므로 객체의 이동에 따른 갱신 처리 속도도 저하된다.

그림 5.1과 그림 5.2는 제안하는 질의 색인에 대해 비트 식별자의 길이와 그룹의 크기 설정이 저장 공간과 처리 속도에 미치는 영향을 나타낸 것이다. 비트 식별자의 크기가 8, 16, 32인 경우 그룹의 크기를 4 ~ 64로 변화하면서 성능을 비교한 결과이다. 비트 식별자의 크기가 저장 공간과 처리속도에 큰 영향을 주지는 않지만 그룹의 크기는 많은 영향을 주고 있다. 저장 공간이나 처리 속도 면에서 그룹 크기가 16일 때 가장 좋은 성능을 보이고 있다.

Collection @ chungbuk

그림 5.1 비트 크기별 처리 속도

그림 5.2 비트 크기별 저장 공간

Collection @ chungbuk

5.1.3 그룹 크기에 따른 성능 비교

그림 5.3과 그림 5.4는 기존에 제안된 CES 기법과 비트 식별자의 크기가 8일 때 제안하는 BID와의 그룹 크기별 저장 공간과 처리속도 비교를 나타낸 것이다. BID 기법은 CES에 비해 적은 저장 공간을 이용하여 색인을 유지하고 그룹 크기가 16인 경우 저장 공간을 가장 적게 이용하는 것이 나타났다. 하지만 그룹 크기가 64인 경우와 같이 그룹 크기가 너무 커져 버리면 오히려 더 많은 저장 공간을 소모하게 된다. 처리 속도 면에는 그룹 크기와 상관없이 BID 기법이 전체적으로 빠른 처리를 수행하였고 CES기법의 경우 그룹 크기가 처리속도에 많은 영향을 미치는 반면 BID기법은 그룹 크기에 큰 영향을 미치지 않는다는 것을 알 수 있다.

Collection @ chungbuk

그림 5.3 그룹 크기별 저장 공간

그림 5.4 그룹 크기별 처리 속도

Collection @ chungbuk

5.1.4 질의 크기에 따른 성능 비교

그림 5.5와 그림 5.6는 기존 CES와 제안하는 BID에 대해 비트 식별자를 8비트로 할 때 최대 질의 크기별 저장 공간과 처리 속도 비교를 나타낸다. 질의 크기가 증가될수록 저장되는 정보가 증가되고 질의 처리를 수행하는데 비교되는 정보의 개수가 증가되기 때문에 처리 속도는 저하된다. 성능 평가 결과 두 기법 모두 질의 크기에 영향을 받지만 BID기법이 CES기법에 비하여 모든 경우에 우수한 성능을 나타낸다.

Collection @ chungbuk

그림 5.5 최대 질의 크기별 저장 공간

그림 5.6 질의 크기별 처리 속도

Collection @ chungbuk

5.1.5 객체의 이동 속도에 따른 성능 비교

그림 5.7은 기존 CES와 제안하는 BID에 대해 비트 식별자를 8비트로 할 때 객체의 속도가 연속 범위 질의 처리에 미치는 영향에 대해서 나타낸 것이다. 제안하는 BID는 보다 적은 연산 비용으로 질의 처리를 수행할 수 있도록 객체가 그룹 안에서 이동을 했을 때 비트 열을 비교를 통해 결과에 영향을 미치는 질의를 찾는다. 따라서 객체가 빠르게 이동 할수록 그룹을 벗어나는 경우가 많이 발생하기 때문에 추가적인 연산 비용이 소모된다. 성능 평가 결과 모든 경우에 BID기법이 CES기법보다 빠른 처리를 수행하였고 객체의 이동 속도에는 BID기법과 CES기법 모두 큰 영향을 받고 있는 것으로 나타났다.

그림 5.7 객체 이동 속도별 처리 속도

Collection @ chungbuk

5.1.6 객체와 질의의 개수에 따른 성능 비교

그림 5.8과 그림 5.9는 질의의 개수와 객체의 개수에 따른 처리 속도를 비교한다. 이 평가는 각 기법의 확장성에 대한 평가를 위해 수행된 것으로서 질의 개수는 1000~16000개 그리고 객체 개수는 4000~64000개를 그리고 객체의 최대 속도 M은 1과 5인 경우를 기준으로 하였다. 모든 경우에 BID기법이 CES기법보다 대략 40%정도의 성능 개선을 보였다.

Collection @ chungbuk

그림 5.8 질의 개수별 처리 속도

그림 5.9 객체 개수별 처리 속도

Collection @ chungbuk

5.1.7 성능 비교 분석

본 논문에서는 다양한 관점에서 제안하는 기법의 성능을 가장 성능이 우수한 CES 기법과 비교하여 하였다, 성능 평가 결과 제안하는 BID 기법이 CES 보다 모든 환경에서 저장 공간 측면과 처리 속도 측면 모두 향상시킨 것을 확인할 수 있었다. 처리 속도의 경우 모든 면에서 CES기법 보다 향상되었음이 확인되었고 그룹 크기에도 큰 영향을 받지 않는 것으로 나타났다. 저장 공간의 경우 그룹의 크기에 영향을 많이 받고 최악의 경우에는 CES 기법 보다 더 많은 저장 공간을 소모한다. 따라서 적용 애플리케이션의 질의 크기와 객체의 속도에 따른 적절한 그룹의 크기 지정은 필수적이다. 결과적으로 소모되는 저장 공간만을 고려하여 그룹의 크기를 지정하면 모든 부분에서 기존의 기법들 보다 나은 성능을 보장한다.

Collection @ chungbuk

5.2 *k*-최근접 질의 처리 기법 성능 비교

5.2.1 평가 환경

본 논문에서 제안하는 거리 관계 패턴을 기반한 *k*-최근접 질의 처리 기법의 특성을 보이기 위해 CPM[16]과 CircularTrip[17] 기법과의 성능 비교를 수행한다. 본 논문에서 제안하는 *k*-최근접 처리 기법은 PB-*k*NN으로 표현한다. 4장에서 돌출된 수식과 성능에 영향을 주는 인자들을 이용하여 시뮬레이션을 해본 결과와 실제 알고리즘 구현에 의한 결과와 동일한 패턴을 나타내었다. 본 논문에서는 알고리즘 구현에 의한 결과들을 이용하여 성능을 평가한다. 성능 평가는 객체의 개수, 질의의 개수, 객체의 이동, 질의의 이동에 따른 질의 처리 속도를 평가하고 또한 저장 공간의 효율성에 대해 평가한다. 연속 질의 평가를 위해 객체의 속도, 이동 비율, 질의의 속도 그리고 이동 비율에 대해서 평가를 수행했다. 구현 및 수행은 IBM xSeries 336, CPU 3.2GHz, RAM 2Gbyte에서 이루어졌다. 제안하는 기법은 PB-*k*NN으로 표기한다. 표 5.2는 성능 평가에서 사용하는 인자들을 나타낸다. 성능 평가에 사용된 데이터는 CPM[16]에서 성능 평가를 위해 사용한 데이터와 동일한 조건으로 생성하였다. *k*-최근접 질의 처리 성능을 평가하기 위해 N_{clr}의 크기는 1에서 10^2까지 변화 시켜 보았다. 그리드 크기는 64^2에서 $1{,}024^2$까지, 객체 개수는 100K에서 1000K까지, 질의 개수는 1K에서 10K까지, *k*값은 1에서 256까지 변화시켜 평가를 수행하였다. 연속 *k*-최근접 질의 처리 성능을 평가하기 위해서 최대 객체/질의 이동 속도를 0.1에서 5까지 변경시켜 보았다. 이동 속도는 0에서 최대 이동 속도 사이까지 범위 내에서 임의로 설정한다. 이때 이동 속도는 셀 한 변의 길이가 1임을 기준으로 설정한 값이다. 그리고 이동 객체 비율과 이동

Collection @ chungbuk

질의 비율을 10%에서 50%까지 변화시켜 평가를 수행하였다.

표 5.2 성능 평가 인자

구분	기본값	설정 범위
N_{clr}	42	12 ~ 102
그리드 크기	512	642 ~ 10242
객체 개수	300K	100K ~ 1000K
질의 개수	5K	1K ~ 10K
k 값	32	1 ~ 256
최대 객체/질의 이동 속도	1	0.1 ~ 5
이동 객체 비율	50%	10% ~ 50%
이동 질의 비율	30%	10% ~ 50%

Collection @ chungbuk

5.2.2 거리 관계 패턴에 따른 성능 변화 분석

k-최근접 질의 처리 시 거리 관계 패턴이 필요로 하는 저장 공간의 크기와 질의 처리 성능의 관계를 조사하였다. 이것은 거리 관계 패턴이 N_{clr}의 값에 따라 최대 오차 E가 결정되기 때문이다. 그림 5.10은 N_{clr}의 증가에 따른 질의 처리 비용과 거리 관계 패턴이 차지하는 저장 공간의 관계를 보여준다. N_{clr}의 값이 4부터 질의 처리 비용은 거의 변화가 없지만 거리 관계 패턴의 크기는 여전히 증가하는 것을 볼 수 있다.

그림 5.10 N_{clr}에 따른 저장 비용과 질의 처리 비용

Collection @ chungbuk

5.2.3 그리드의 크기에 따른 성능 비교

적절한 그리드의 크기를 결정하기 위해 그리드의 조밀도가 질의 처리를 위해 소모되는 연산 비용과 저장 비용에 미치는 영향을 조사하였다. 그림 5.11은 동일한 크기의 공간에 대해서 그리드 색인의 조밀도에 따른 질의 처리 속도의 변화를 나타낸다. 그리드의 크기에 관계없이 PB-*k*NN이 CPM과 CircularTrip 보다 빠른 질의 처리 속도를 보장한다. 그리드 크기가 커질수록 두 알고리즘 모두 질의 처리 속도가 저하되지만, CPM과 CircularTrip의 속도 저하율 보다 PB-*k*NN의 속도 저하율이 작음을 나타낸다. 그리드의 조밀도가 커짐에 따라 CPM의 경우 근접한 셀을 찾기 위해 사용하는 우선순위 큐로 인해 비롯되는 연산 비용이 커지게 된다. 하지만 PB-*k*NN은 단순히 CLR을 순회하면서 질의를 처리하기 때문에 CPM보다는 그리드의 조밀도에 따른 영향을 적게 받는다. 그리드 크기가 64^2~256^2인 경우 까지는 속도 차이가 15~25%정도로 증가하며 그 이후부터는 보다 큰 폭으로 벌어지는 것을 확인할 수 있다. 그리드 크기가 약 256^2인 지점에서 'U'형태로 처리율의 변화가 일어난다. 한셀 당 포함하는 객체의 수가 증가하면 질의를 처리하기 위해 셀을 탐색할 때마다 그 셀의 객체와 질의 점간의 거리연산 비용도 증가하게 된다. 그래서 객체의 수에 비해 셀의 조밀도가 작은 경우 오히려 연산 비용이 증가하게 된다. 이러한 현상은 식 4.23에서 나타내고 있으며, 최적의 성능을 위해서는 객체의 수를 고려하여 적절한 조밀도로 그리드 크기를 결정해야 한다.

Collection @ chungbuk

그림 5.12는 그리드 크기에 따른 저장 비용의 변화를 나타낸다. BASE는 질의 테이블과 거리 관계 패턴을 제외한 기본 저장 비용을 나타낸다. 다시 말해서 BASE는 객체 리스트와 질의 리스트를 포함하는 그리드 정보와 객체 테이블 정보를 유지하는데 요구되는 저장 공간의 크기를 나타낸다. BASE에 해당하는 부분은 질의 처리 기법에 관계없이 연속 질의 처리를 위해 반드시 유지되어야 하는 정보이다. 나머지 자료구조는 질의 처기 기법의 특성에 따라 달라진다. CPM의 경우 질의 테이블에 우선순위 큐와 셀 방문 리스트를 유지하여 연속 질의를 처리한다. CircularTrip의 경우 질의 테이블에 셀 방문 리스트를 유지한다. PB-*k*NN의 경우 CPM이나 CircularTrip이 질의 테이블에 유지하는 우선순위 큐와 셀 방문 리스트를 유지하지 않고, 모든 질의가 공유해서 사용하는 거리 관계 패턴을 사용한다. PB-*k*NN은 모든 질의가 공유하는 거리 관계 패턴을 이용하기 때문에 셀의 조밀도에 따라 BASE에서 추가적으로 요구되는 저장 공간이 일정하지만, CPM과 CircularTrip의 경우 조밀도가 커지면 방문하는 셀의 개수가 늘어나기 때문에 각 질의에서 유지하는 정보량이 크게 증가한다. 기존의 기법과 PB-*k*NN의 비용차이는 식 4.13에서 보여주는 것과 동일하다. 저장 공간의 차이는 그리드 크기에 비례하여 증가하며, 그리드 크기가 64^2에서 512^2로 변화하는 동안 PB-*k*NN이 다른 기법들에 비해 약 10%에서 35%의 저장 공간 소모율의 감소를 보여준다. 1024^2부터 그 차이는 60% 이상이 된다. 결론적으로 CPM과 CircularTrip이 소모하는 저장 비용에 비해 PB-*k*NN가 소모하는 저장비용은 매우 작고, 확장성이 있다는 것을 나타낸다.

Collection @ chungbuk

그림 5.11 그리드 크기에 따른 질의 처리 속도의 변화

그림 5.12 그리드 크기에 따른 저장 비용

Collection @ chungbuk

5.2.4 질의와 객체 개수에 따른 성능 비교

그림 5.13은 객체의 개수에 따른 질의 처리 시간의 변화를 나타낸다. 객체의 밀도가 증가하면 *k*-최근접 객체를 찾기 위해 탐색해야 하는 셀의 개수가 감소하기 때문에 근접한 셀을 찾기 위한 연산이 감소한다. 그래서 객체수가 증가하면 질의 처리 시간이 감소한다. 이러한 현상은 CPM, CircularTrip 그리고 PB-*k*NN에서 모두 발생한다. 그림 21에서 객체 개수가 50K에서 500K개로 변화하는 동안, CPM의 질의 처리 시간은 약 2,700에서 270으로 감소하고 PB-*k*NN의 질의 처리 시간도 약 500에서 200으로 감소하는 것을 보여준다. 객체 수에 따른 PB-*k*NN의 질의 처리 속도는 다른 기법들에 비해서 적게는 1.3배에서 많게는 5배 이상이 된다. 전체적으로 PB-*k*NN의 질의 처리 속도가 CPM과 CircularTrip보다 객체의 밀도에 관계없이 모든 상황에서 빠르다는 것을 확인 시켜준다.

그림 5.13 객체 개수에 따른 질의 처리 시간 변화

Collection @ chungbuk

그림 5.14는 질의 개수에 따른 질의 처리 시간의 변화를 나타낸다. 질의 개수가 증가는 저장 공간의 증가를 초래할 뿐만 아니라 질의 처리 시간도 증가한다. 질의 개수가 증가하기 때문에 발생하는 당연한 결과이다. 본 논문의 실험환경에서는 PB-*k*NN의 질의 개수에 상관없이 CPM과 CircularTrip보다 적은 연산 비용을 소모하고, 약 30% 정도 질의 처리 시간을 단축하였다.

그림 5.14 질의 개수에 따른 질의 처리 시간 변화

Collection @ chungbuk

5.2.5 *k*-값에 따른 질의 처리 시간 비교

그림 5.15은 *k*값에 따른 질의 처리 시간의 변화를 나타낸다. *k*값을 1~256까지 변경하며 평가하였다. *k*값이 증가함에 따라 성능의 차이가 커짐을 발견할 수 있다. *k*값의 증가는 결국 탐색되는 셀의 개수를 증가시키기 때문이다. 두 기법 모두 *k*값이 작을 때는 거의 비슷한 시간 변화를 가지지만, 그 값이 커짐에 따라 시간 변화의 차이가 커지게 된다. 그림 23에서는 *k*값이 증가함에 따라 질의 처리 시간의 변화 비율은 CPM과 CircularTrip이 PB-*k*NN보다 약 4배 정도로 급격하게 변환되는 것을 보여준다. 결론적으로 PB-*k*NN이 CPM에 비해 *k*값의 증가에 따른 연산 비용의 증가율을 감소시켰다.

그림 5.15 *k* 값에 따른 질의 처리 시간 변화

Collection @ chungbuk

5.2.6 객체 이동 속도에 따른 질의 처리 시간 비교

그림 5.16는 최대 객체 이동 속도에 따른 연속 질의 처리 시간의 변화를 나타낸다. 이동 속도는 각 차원 별(유클리드 공간에서 x, y) 최대 속도를 나타낸다. 객체가 이동하는 속도에 따라 재 수행 되어야 하는 질의의 수가 증가한다. 때문에 객체의 이동 속도가 증가하면 연속 질의 처리 비용이 증가한다. 그림 24에서 객체의 이동 속도가 작을 때는 PB-*k*NN이 CPM과 CircularTrip에 비해 약 3배 정도 빠른 질의 처리시간을 보이지만 객체의 이동 속도가 증가함에 따라서 그 차이는 약 5배까지 증가한다.

그림 5.16 객체 이동 속도에 따른 연속 질의 처리 시간 변화

Collection @ chungbuk

5.2.7 질의 이동 속도에 따른 질의 처리 시간 비교

질의 이동 속도에 따른 질의 처리 시간의 변화를 비교하여 질의 이동에 따른 연속질의 처리 성능을 평가해 보았다. CPM과 CircularTrip은 질의가 이동하면 무조건 질의를 초기화 하여 처음부터 수행하기 때문에 질의 이동속도에 따른 비용 변화가 발생하지 않는다. 반면에 PB-*k*NN은 질의가 한 VDC내에서 이동한 경우 점진적인 질의 수행을 하기 때문에 질의 이동 속도에 따른 비용 변화가 발생한다. 그림 5.17에서 CPM과 CircularTrip은 질의 이동 속도에 상관없이 항상 동일한 질의 처리 비용을 유지하지만, PB-*k*NN의 경우 이동속도의 변화에 따라 질의 처리 비용이 변하는 것을 보여준다. PB-*k*NN의 경우 질의의 최대 이동 속도가 0.1인 경우 다른 기법들에 비해 약 2.3배 정도 빠른 질의 처리 속도를 보여주고, 질의 이동 속도가 증가함에 따라서 약 1.6배 정도 빠른 질의 처리 속도를 보여준다.

그림 5.17 질의 이동 속도에 따른 질의 처리 속도

Collection @ chungbuk

5.2.8 객체 이동 량에 따른 질의 처리 시간 비교

그림 5.18은 객체 이동 량에 따른 질의 처리 속도를 나타낸다. 객체가 동시에 이동하는 량이 증가할수록 재 수행해야 하는 질의 개수가 증가하므로 연속 질의 처리 비용의 증가를 초래한다. 각 객체의 이동에 대해서 질의 처리가 수행되는 것이 아니고 동시에 이동한 객체들에 대해서 한 번에 질의가 수행되기 때문에 이동하는 객체의 수의 증가에 따라 연속 질의 처리 비용의 증가량이 감소하는 현상이 나타난다. 이러한 현상은 두 알고리즘에서 동일하게 나타난다. 객체 이동 량에 따른 질의 처리 속도는 객체 이동 량이 10%인 경우, PB-*k*NN이 CPM과 CircularTrip에 비해 약 3배 정도 되고, 객체 이동 량이 증가하면서 5배 이상 차이가 난다.

그림 5.18 객체 이동 량에 따른 질의 처리 시간 변화

Collection @ chungbuk

5.2.9 질의 이동 량에 따른 질의 처리 시간 비교

그림 5.19은 질의 이동 량에 따른 질의 처리 시간의 변화를 나타낸다. 질의 이동 량이 증가함에 따라 연속 질의 처리 비용도 증가한다. 두 알고리즘 모두 질의 개수의 증가에 따른 효율성에 대해 고려하지 않기 때문에 특징적인 부분은 발행하지 않는다. 그림 5.19에서는 질의의 이동 량에 관계없이 PB-*k*NN이 CPM과 CircularTrip보다 약 30% ~ 25%정도 적은 질의 처리 비용을 소모하는 것을 보여준다.

그림 5.19 질의 이동 량에 따른 질의 처리 시간 변화

Collection @ chungbuk

VI. 결론

본 논문에서는 이동 객체를 대상으로 하는 연속 영역 질의 처리를 위해 기존에 제안된 여러 기법들의 단점을 자세히 분석하고, 기존 기법에 비해 보다 적은 공간을 사용하면서 빠른 연속 질의를 처리하기 위한 질의 처리 기법을 제안하였다. 연속 질의 처리는 객체의 갱신에 의해 질의 결과의 변경을 처리하기 위해 영향을 받는 질의를 빠르게 알아내는 것이 중요하다. 제안하는 연속 질의 처리 기법은 질의 색인 기법을 위해 메인 메모리 기반의 그리드 구조를 사용하고 그리드로 분할된 셀들은 그룹 단위로 관리한다. 본 논문에서는 질의를 나타내기 위해 비트 식별자를 이용하여 저장 공간의 효율과 빠른 처리 속도 제공하고 각 셀에서 유지하는 질의와의 포함관계를 나타내는 비트 열을 이용하여 객체의 위치 갱신이 영향을 주는 질의를 빠르게 파악할 수 있다.

효율적인 연속 k-최근접 질의 처리를 위해 그리드 환경에서의 거리 관계 패턴을 제안하였다. 또한 제안한 거리 관계 패턴을 기반으로 하는 k-최근접 질의 처리 기법을 제안하였다. 동일한 모양과 크기를 갖는 셀들이 동일한 간격으로 배치되어 있을 경우 제안하는 거리 관계 패턴은 적은 연산 비용으로 근접한 셀들을 순서대로 탐색해 질의 처리 비용을 획기적으로 감소시켰고, 저장 비용의 낭비를 방지하였다. 또한 수식을 이용하여 PB-kNN과 CPM의 질의 처리 비용과 저장 비용을 분석하였다. 마지막으로, 대표적인 k-최근접 질의 처리 기법인 CPM과의 성능 비교를 통해, 저장 비용과 처리 비용 측면에서 제안하는 k-최근접 질의 처리 기법의 성능이 모두 우수함을 확인하였다.

Collection @ chungbuk

참고문헌

[1] D. L. Lee, J. Xu, B. Zheng, and W. C. Lee. "Data Management in Location-Dependent Information Services," IEEE Pervasive Computing, Vol. 1, No. 3, pp. 65-72, 2002.

[2] J. Xu, B. Zheng, M. Zhu, and D. L. Lee. "Research Challenges in Information Access and Dissemination in a Mobile Environment," Proc. Pan-Yellow-Sea International Workshop on Information Technologies for Network Era, pp. 1-8, 2002.

[3] H. Hu, J. Xu, and D. L. Lee. "A Generic framework for Monitoring Continuous Spatial Queries over Moving Objects," Proc. ACM SIGMOD International Conference on Management of Data, pp. 479-490, 2005.

[4] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. "Continuous Query Processing of Spatio-Temporal Data Streams in PLACE," Journal of Geoinformatica, Vol. 9, No. 4, pp. 343-365, 2005.

[5] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. "Main Memory Evaluation of Monitoring Queries Over Moving Objects," Distributed and Parallel Databases, Vol. 15, No. 2, pp. 117-135, 2004.

[6] K. L. Wu, S. K. Chen, and P. S. Yu. "Processing continual range queries over moving objects using VCR-based query indexes," Proc. IEEE International Conference on Mobile and Ubiquitous Systems : Networking and Services, pp. 226-235, 2004.

[7] K. L. Wu, S. K. Chen, and P. S. Yu. "On Incremental Processing of Continual Range Queries for Location-Aware Services and Applications," Proc. Annual International Conference on Mobile and Ubiquitous Systems, pp. 261-269, 2005.

Collection @ chungbuk

[8] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. "Query Indexing and Velocity Constrained Indexing : Scalable Techniques for Continuous Queries on Moving Objects," IEEE Transactions on Computers, Vol. 51, No. 10, pp. 1124-1140, 2002.

[9] X. Wang, Q. Zhang, W. Sun, W. Wang, and B. Shi. "cGridex: Efficient Processing of Continuous Range Queries over Moving Objects," Proc. International Conference on Advances in Web-Age Information Management, pp. 345-356, 2005.

[10] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases," Proc. ACM SIGMOD International Conference on Management of Data, pp. 623-634, 2004.

[11] B. Gedik, K. L. Wu, P. S. Yu, and L. Liu. "Motion adaptive indexing for moving continual queries over moving objects," Proc. ACM International conference on Information and Knowledge Management, pp. 427-436, 2004.

[12] A. Gutmann, "R-trees: A dynamic index structure for spatial searching", In Proc. ACM SIGMOD International Conference on Management of Data, 1984.

[13] Xiong, X., Mokbel, M., Aref, W. "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases," Proc. IEEE International Conference on Data Engineering (ICDE), 2005.

[14] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao, "A Threshold-Based Algorithm for Continuous Monitoring of k Nearest Neighbors," IEEE Transactions on Knowledge and Data Engineering, vol. 17 no. 11, pp. 1451-1464, November 2005

[15] X. Yu, K. Pu, and N. Kouda16s, "Monitoring *k*-nearest Neighbor Queries over Moving Objects," Proc. International Conference on Data Engineering, pp.

Collection @ chungbuk

631-642, 2005.

[16] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias, "Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring," Proc. ACM Conference on Management of Data, pp. 634-645, 2005.

[17] M. A. Cheema, Y. Yuan, and X. Lin, "CircularTrip: An Effective Algorithm for Continuous *k*NN Queries," Proc. International Conference on Database systems for Advanced Applications (DASFAA), 2007.

[18] Z. Song, and N. Roussopoulos, "*K*-nearest neighbor search for moving query point," Proc. Symposium on Spatial and Temporal Databases (SSTD), 2001.

[19] Y. Tao, and D. Papadias, "Time-parameterized queries in spatio-temporal databases," Proc. ACM SIGMOD international conference on Management of data, 2002.

[20] Y. Tao, and D. Papadias, "Continuous nearest neighbor search," Journal of Very Large Database (VLDB), 2002.

[21] J. Zhang, M. Zhu, D. Papadias, Y. Tao, D.L. Lee, "Location-based spatial queries," Proc. ACM SIGMOD international conference on Management of data, 2003.

[22] T. Brinkhoff, "A framework for generating network-based moving objects," Proc. GeoInformatica, Vol. 6, No. 2, pp. 153-180, 2002.

[23] P. K. Agarwal, L. Arge, and J. Erickson. "Indexing moving objects," Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp.175-186, 2000.

[24] G. Kollios, D. Gunopulos, and V. J. Tsotras. "On indexing mobile objects," Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp.261-272, 1999.

Collection @ chungbuk

[25] D. Pfoser, C. S. Jensen, and Y. Theodoridis. "Novel approaches to the indexing of moving object trajectories," Proc. International Conference on Very Large Data Bases. pp.395-406, 2000.

[26] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. "Indexing the positions of continuously moving objects," Proc. ACM SIGMOD International Conference on Management of Data, pp.331-342, 2000.

[27] C. S. Ensen, D. Lin, and B. C. Ooi, "Query and Update Efficient B^+-Tree Based Indexing of Moving Objects," Proc. International Conference on Very Large Database(VLDB), pp. 768-779, 2004.

[28] M. L. Yiu, Y. Tao, and N. Mamoulis, "The Bdual-tree: Indexing Moving Objects by SpaceFilling Curves in the Dual Space," Journal of Very Large Database(VLDB), 2006.

[29] S. Chen, B. C. Ooi, K. L. Tan, and M. A. Nascimento, "ST^2B-tree: A Self-Tunable Spatio-Temporal B+-tree Index for Moving Objects," Proc. ACM SIGMOD International Conference on Management of data, 2008.

[30] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R^*-tree: an efficient and robust access method for points and rectangles," Proc. ACM SIGMOD International Conference on Management of data, 1990.

[31] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A dynamic index for multi-dimensional objects," Journal of Very Large Database(VLDB), 1987.

[32] D. A. White, and R. Jain, "Similarity indexing with the SS-tree," Proc. International Conference on Data Engineering, 1996.

[33] N. Katayama, and S. Satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," Proc. ACM SIGMOD International Conference on Management of Data, 1997.

[34] K. Lin, H. V. Jagadish, and C. Faloutsos, "The TV-tree: An Index Structure for High-Dimensional Data", Journal of Very Large Database(VLDB), vol. 3, pp. 517-542, 1995.

Collection @ chungbuk

[35] S. Berchtold, D. A. Keim and H.-P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," Proc. International Conference on Very Large Database(VLDB), 1996.

[36] 송석일, 이석희, 조기형, 그리고 유재수, "VA-Tree: 대용량 데이터를 위한 효율적인 다차원 색인구조," 한국 멀티미디어학회논문지, vol.6, no.5. pp. 753-768. 2003.

[37] M. A. Nascimento, and J. R. O. Silva, "Towards historical R-trees," Proc. ACM Symposium on Applied Computing, SAC, 1998.

[38] Y. Tao, and D. Papadias, "Efficient Historical R-trees," Proc. International Conference on Scientific and Statistical Database Management (SSDBM), 2001.

[39] Y. Park, D. Seo, H. Park, and J. Yoo, "An Index Structure for Efficient *k*-NN Query Processing in Location Based Services", Proc. International Conference on Ubiquitous Information Technologies & Applications, pp. 165-170, 2009.

Collection @ chungbuk

감사의 글

엊그제 연구실에 들어와 공부를 시작한 것 같은데 벌써 6년간의 공부를 마쳤습니다. 앞만 보고 달려와 놓친 것, 잃은 것도 많은 시간이었지만 그 만큼 제 인생 전체를 볼 때 가장 큰 발전과 발돋움의 기간이었음을 확신합니다. 그동안 너무나 부족했던 저에게 도움을 주신 모든 분들께 감사의 마음을 전하고 싶습니다.

먼저 무엇이든 할 수 있다는 용기를 주시고, 좋은 결과가 아닌 노력하는 모습을 높게 평가해 주시고, 아낌없는 지원과 믿음으로 지켜봐 주신 그리고 항상 사랑으로서 보살펴 주신 세상에서 가장 존경하는 부모님께 감사드립니다. 그리고 옆에서 옳은 길을 걸을 수 있게 조언해주고, 물질적으로 그리고 정신적으로 큰 도움이 된 상호형, 원영누나, 신영누나, 주영누나, 은경누나 그리고 형수님과 매형들에게 감사드립니다.

매번 바쁘다며 이래저래 만나지도 못하고, 놀지도 못했지만, 매번 잊지 않고 연락해 주고, 살아오는데 동반자가 되어준 친구들에게 감사를 드립니다. 남원에 가면 항상 반겨주고, 잘못 한 일이 있으면 지적해주고, 자주 연락을 하며 관심을 가져준 혁이, 직장도 있고, 결혼도 한, 나의 부러움의 대상인 배균이, 식당을 경영하며 알콩달콩 살아가는 태현이와 수정이, 진로 상담 해주느라 고생이 많았던 지숙이, 벌써 둘째를 낳아 기르고 있는 진호에게 감사를 드립니다.

후배로서 부족한 모습이었음에도 불구하고, 선배로서의 역할을 충실히 해주고, 좋은 방향으로 이끌어준 연구실 선배들에게 깊은 감사를 드립니다. 부족한 모습이지만 외부에서 지켜봐주신 최길성, 이석희, 강형일, 신재룡, 손충범 박사님들께 감사를 드립니다. 학문적 그리고 정신적 지주의 역할을 해준 석일형, 지난 6년 동안 연구에 관심을 가져주고 연구의 기틀을 마련해준 경수형, 가끔 의사소통이 잘되지 않아서 대화가 힘들 때가 있지만, 순수하고 열정적으로 모

Collection @ chungbuk

든 일에 임하는 모습을 가진 수민형, 프로젝트를 같이 하면서 저 때문에 가장 마음고생을 많이 한 그리고 카리스마 있는 태호형, 졸업했지만 여전히 후배들에게 많은 관심을 가져주고 충고도 따뜻하게 해주는 영수형, 만나면 항상 즐거운 얘기로 시간 가는 것을 잊어버리게 하는 석재형, 힘든 일이 있을 때, 랩 매니저로서의 고민을 들어주고 용기를 준 동민형, 후배지만 형으로 배려해주고 친구가 되어준 종현, 인생을 열정적으로 살아가는 명호에게 감사를 드립니다.

또한 부족한 모습임에도 선배로 대우 해준 연구실 후배들에게 깊은 감사를 드립니다. 만날 때마다 사회에 대한 다양한 충고와 조언을 아끼지 않은 그리고 사업에 뛰어든 현호(박현호), 선배로서 부족하지만 잘 따라주고, 덩치만큼이나 큰 그릇을 가진 그리고 순수한 마음을 가진 준호, 늘 조용히 혼자서 열심히 공부하는 이쁜 지희, 외국인 학생으로서 언어적 문제를 극복하고 이제는 연구의 길로 들어선 Li He, 조용하고 부지런하고 맡은 일에 최선을 다하는 효준, 맨날 책상에서 잠만 자는 듯 하지만 할 것 다 해놓는 그리고 내 운전 문하생 임실장 종태, 연애 뿐 만 아니라 모든 일에도 적극적인 광구, 옆에서 얘기도 많이 들어주고 객관적인 평가도 매섭게 해준 진주, 개인 적으로 진보상을 주고 싶은 책임감 있고 착한 용진, 맡은 일을 꾸준히 하려고 노력하고 밤 시간을 함께 해준 꽃미남 현호(권현호), 작고 귀여운 그리고 고집쟁이 포미미, 모든 일을 즐겁게 하려고 노력하지만 가끔 안 돼서 투덜대는 모습이 귀여운 미경이, 술 잘 마시고 놀기 좋아하고 일도 열심히 하는 그리고 모험을 즐기는 미림, 옆에서 이것저것 보조 해주고 게으름과 지저분한 모습도 잘 참아주고 승짱을 좋아하는 쾌활하고 명랑한 김비서 현주, 헌혈도 많이 하고 봉사 활동도 많이 하고 선배도 잘 챙기고 그리고 눈치도 빠른 순수한 소년 수(강수), 다른 학생들 보다 배경 지식이 없다는 우려만큼이나 더욱 열심히 하려고 노력하는 은주, 조교 경력 때문인지 외모답지 않게 터프한 현정, 학부생인데도 대학원에 들어와서 열심히 공부하는 탄과 동교, 그리고 힘들 때 용기가 되어주고

Collection @ chungbuk

힘을 실어준 그리고 동기로서 박사과정의 시작과 끝을 함께한 동욱에게 감사를 드립니다.

지난 6년 동안 늘 아낌없는 격려와 지도로 저를 이끌어 주시고, 능력 보다 인성의 중요성을 일깨워 주신 조기형 교수님과 유재수 교수님께 깊은 감사를 드립니다. 호원대학교에서 건축공학전공으로 입학한 제게 이러한 길을 걸을 수 있게 기회를 만들어 주신 최영희 교수님께 깊은 감사를 드립니다.

마지막으로 오늘이 있기까지 옆에서 지켜봐주고 인내해준 사랑하는 희진에게 감사를 드립니다.

오늘의 작은 열매에 자만하지 않고 계속 정진하면서 살아가도록 하겠습니다.

Collection @ chungbuk