시간지원 데이터를 위한 분리 저장 구조와 데이터 이동 방법

윤 홍 원† · 김 경 석††

요 약

시간지원 데이터 모델과 시간지원 질의어에 관한 연구가 활발한 반면에 시간지원 데이터의 특성을 고려한 저장 방법과 데이터 이동에 관한 연구는 많지 않다. 이 논문에서는 시간지원 데이터의 시간적 특성을 고려해서 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트로 분리해서 저장하는 P-C-F 저장 구조를 제안하였다. 또한, P-C-F 저장 구조를 기반으로 하는 데이터 이동 방법 두 가지를 제안하였다.

이 논문에서 제안한 P-C-F 저장 구조에서는 데이터 이동 프로세서가 데이터를 이동하기 위한 시기를 결정하는 방법과 데이터를 이동하는 과정을 제안하였다. 이 논문에서 제안한 시간단위에 의한 이동 방법에서는 각 세그먼트에 저장되는 개체 버전을 정의하였고, 각 세그먼트에서 데이터의 유효성을 검사하고 이동하는 과정을 보였다. LST-GET에 의한 이동 방법에서는 LST-GET를 정의하였고, LST와 GET를 기준으로 각 세그먼트에 저장되는 개체 버전과 이동 대상이 되는 개체 버전을 검색하고 이동하는 과정을 보였다. 모의 실험을 통해서 기존의 저장 구조와 P-C-F 저장 구조에 대해서 질의에 대한 성능을 분석하였고, 데이터 이동 방법들의 성능을 분석하였다.

A Time-Segmented Storage Structure and Migration Strategies for Temporal Data

Hong-Won Yun† · Kyong-Sok Kim††

ABSTRACT

Numerous proposals for extending the relational data model as well as conceptual and object-oriented data models have been suggested. However, there has been relatively less research in the area of defining segmented storage structure and data migration strategies for temporal data. This paper presents the segmented storage structure in order to increment search performance and the two data migration strategies for segmented storage structure.

This paper presents the two data migration strategies : the migration strategy by Time Granularity, the migration strategy by LST-GET. In the migration strategy by Time Granularity, the dividing time point to assign the entity versions to the past segment, the current segment, and future segment is defined and the searching and moving process for data validity at a granularity level are described. In the migration strategy by LST-GET, we describe the process how to compute the value of dividing criterion. Searching and moving processes are described for migration on the future segment and the current segment and entity versions to assign on each segment are defined. We simulate the search performance of the segmented storage structure in order to compare it with conventional storage structure in relational database system. And extensive simulation studies are performed in order to compare the search performance of the migration strategies with the segmented storage structure.

† 정 회 원 : 신라대학교 컴퓨터정보공학부 교수
†† 정 회 원 : 부산대학교 전자계산학과 교수
논문접수 : 1998년 11월 14일, 심사완료 : 1999년 1월 29일

1. 서 론

시간지원 데이터베이스(Temporal Database)는 시간에 따라 변하는 객체의 정보를 기록하고 객체의 현재 상태, 과거 상태, 그리고 계획된 미래 상태에 대해서 질의할 수 있는 특징을 가지고 있다[12,22,23]. 지난 20여년 동안 진행되고 있는 시간지원 데이터베이스에 관한 연구는 시간지원 데이터 모델(temporal data model)과 시간지원 질의어(temporal query language)[3,6,9,14,17,18,19,20,21]에 집중되었으며, 최근에는 여러 가지 시간지원 인덱스 구조[4,11,12,13,15]와 시간지원 데이터베이스 프로토타입 시스템이 나오고 있다[2].

시간지원 데이터 모델과 시간지원 질의어에 관한 연구가 활발한 반면에, 시간지원 데이터(temporal data)의 특성을 고려한 저장 방법과 데이터 이동(data migration)에 관한 연구는 많지 않다. 지금까지 구현된 대부분의 시간지원 데이터베이스 시스템은 기존의 상업용 데이터베이스 관리 시스템 위에 시간지원 질의어를 처리하는 전처리기를 두는 수준에서 개발되었으며, 기존의 데이터를 저장하는 방법대로 시간지원 데이터를 저장하고 있다[2]. 이러한 기존의 데이터 저장 방법에서는 시간지원 데이터의 시간적 특성을 고려하지 않고 있다.

시간지원 데이터는 과거 특정한 기간에 유효했던 개체 버전, 현재 유효한 개체 버전, 그리고 미래의 특정 기간 동안 유효한 개체 버전으로 나눌 수 있다. 지금까지 시간지원 데이터베이스에서 다루어 온 시간지원 데이터는 과거에 유효했던 개체 버전과 현재 유효한 개체 버전이 대부분이었으나, 최근에는 미래의 특정 기간 동안 유효한 데이터를 고려한 연구 결과가 나오고 있다[5,8,10,23,24,25,26,27].

시간지원 데이터의 시간적 특성을 고려하면 과거에 유효했던 개체 버전은 과거 세그먼트에 저장하고, 현재 유효한 개체 버전은 현재 세그먼트에 저장하고, 그리고 미래에 유효한 개체 버전은 미래 세그먼트에 저장할 수 있다. 시간지원 데이터를 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트로 분리한 저장 구조에서는 시간지원 질의에 대해서 빠른 응답을 보일 뿐만 아니라, 오래된 개체 버전을 적절한 저장 매체에 저장할 수 있으므로 저장 비용의 측면에서 효율적이다.

이 논문에서는 시간지원 데이터의 시간적 특성을 고려해서 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트로 분리해서 저장하는 분리 저장 구조를 제안한다. 또한, 이 논문에서 제안하는 분리 저장 구조를 기반으로 하는 데이터의 이동 방법 두 가지를 제안한다. 모의 실험을 통해서 이 논문에서 제안하는 데이터 이동 방법들의 성능을 분석한다.

이 논문의 구성은 다음과 같다. 2장에서는 시간지원 데이터의 분리 저장 구조와 데이터 이동과 관련된 기존의 연구에 대해서 살펴본다. 3장에서는 이 논문에서 제안하는 시간지원 데이터의 분리 저장 구조인 P-C-F (Past-Current-Future) 저장 구조를 보이고, P-C-F 저장 구조를 기반으로 하는 데이터 이동 방법을 제안한다. 4장에서는 기존의 관계형 데이터베이스 시스템의 저장 구조와 이 논문에서 제안한 분리 저장 구조에 대해서 질의에 대한 성능을 비교한다. 또한, 이 논문에서 제안한 시간단위에 의한 이동 방법과 LST-GET(Least valid Start Time-Greatest valid End Time)에 의한 이동 방법에 대해서 질의에 대한 성능을 평가한다. 마지막으로, 5장에서는 결론과 향후 연구 과제에 대해서 살펴본다.

2. 관련 연구

2장에서는 기존의 시간지원 데이터베이스와 관련된 저장 구조와 분리 저장 방법에 대해서 살펴본다. 또한, 시간지원 데이터의 이동에 관한 기존의 연구에 대해서 살펴본다.

2.1 기존의 저장 구조와 분리 저장 방법

Ahn과 Snodgrass는 시간지원 데이터에 대한 검색 속도를 향상시키고 공간을 효율적으로 활용하기 위해서 시간지원 데이터의 저장 장소를 현재 저장소와 이력 저장소로 나누고, 이력 저장소의 구조에 대해서 역 체인, 접근 리스트, 클러스터링, 스택, 그리고 셀루러 체인 등 다섯 가지 저장 구조를 제안하였다[1].

역 체인 구조는 현재 저장소에 저장된 현재 버전을 시작으로 해서 연관된 이력 버전과 연결되는데, 현재 버전은 이력 저장소에 있는 가장 최근의 이력 버전으로 연결되고, 나머지 연관된 이력 버전은 가장 최근의 이력 버전으로부터 시간의 역순으로 연결된다. 체인을 따라가면서 탐색 속도가 느려지는 역 체인 구조의 단점을 보완하기 위해서, 접근 리스트 구조에서는 현재 저장소와 과거 저장소 사이에 현재 버전과 연관된 이력 버전의 포인터를 저장하는 접근 리스트 포인터를

두어서 작은 인덱스 역할을 하도록 하고 있다. 접근 리스트 구조는 이력 버전이 이력 저장소 안에서 여러 블록에 분산되어 저장되는 경향이 있다. 따라서 어떤 현재 버전과 연관된 이력 버전을 찾을 때 디스크의 접근 회수를 증가하게 된다. 클러스터링 구조는 현재 버전과 연관된 이력 버전을 연속적인 블록에 저장한다.

스택 구조는 일정한 개수의 이력 버전만 저장할 수 있는 공간을 할당하고 가장 최근의 이력 버전을 중심으로 저장한다. 이력 버전이 할당된 일정한 개수를 넘으면 가장 오래된 이력 버전은 잃게 된다. 셀룰러 체인 구조는 한 셀 안에 현재 버전과 연관된 다수의 이력 버전을 저장하고, 셀이 넘치면 연관된 셀들을 역 체인 구조와 같은 방법으로 연결한다. 셀룰러 체인 구조는 역 체인과 스택을 결합한 구조로 볼 수 있다.

Ahn과 Snodgrass가 제안한 저장 구조들은 모두 과거에 유효했던 개체 버전과 현재 유효한 개체 버전만을 고려하고 있으며 미래에 대한 계획이나 예측 등으로 인해서 발생할 수 있는 미래에 유효한 개체 버전은 고려하지 않았다. 이 논문에서는 과거에 유효했던 개체 버전과 현재 유효한 개체 버전은 물론, 미래의 특정 기간 동안에 유효한 미래의 개체 버전을 고려한 분리 저장 구조를 제안한다.

Sarda는 HDBMS(Historical DBMS)가 수행할 역할로써 이력 릴레이션(historical relation)을 과거, 현재, 그리고 미래 세그먼트로 분리해서 관리하는 방안을 개념적으로 제안하였다[16]. Sarda의 제안에 의하며, 과거 세그먼트에는 유효 끝 시각이 현재 시각보다 작은 개체 버전을 저장하고, 현재 세그먼트에는 유효 시작 시각이 현재 시각보다 작거나 같고 유효 끝 시각이 현재 시각보다 크거나 같은 개체 버전을 저장한다. 또한, 미래 세그먼트에는 유효 시작 시각이 현재 시각보다 큰 개체 버전을 저장한다.

Sarda는 시간지원 데이터에 과거 데이터, 현재 데이터뿐만 아니라, 개인이나 조직체의 미래에 대한 계획을 나타내는 미래 데이터를 포함시키고 있다. Sarda는 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트에 저장될 개체 버전을 분리하기 위해서 현재 시각을 사용하는 개념적인 방안을 제안하였다. 그러나, 과거 세그먼트, 현재 현재 세그먼트, 그리고 미래 세그먼트의 분리 구조를 관리하는 구체적인 방법은 언급하지 않았다.

Kouramajian은 [12]에서 시간지원 데이터의 저장 장소를 광 디스크와 자기 디스크로 구분해서 오래된 과거에 유효했던 개체 버전은 광 디스크에 저장하고 가까운 과거에 유효했던 개체 버전과 현재 유효한 개체 버전은 자기 디스크에 저장한다. Kouramajian은 광 디스크과 자기 디스크에 저장될 개체 버전을 구분하기 위해서 현재 유효한 개체 버전 중에서 가장 작은 유효 시작 시각을 경계값으로 쓴다.

경계값보다 작은 유효 끝 시각을 가지는 개체 버전은 대부분 광 디스크에 저장하고 참조한다. 유효 시작 시각이 경계값보다 작고 유효 끝 시각이 경계값보다 큰 개체 버전은 광 디스크와 자기 디스크 양쪽 모두에 저장하고 참조한다. 또한, 유효 시작 시각이 경계값보다 큰 개체 버전은 자기 디스크에 저장하고 참조한다.

Kouramajian은 시간지원 데이터를 분리하면서 과거에 유효했던 개체 버전과 현재 유효한 개체 버전만을 고려하였으며 미래에 유효한 개체 버전은 다루지 않았다. 이 논문에서는 과거에 유효했던 개체 버전, 현재 유효한 개체 버전을 포함해서 미래에 유효한 개체 버전을 분리 저장 구조에서 관리하는 방법을 제안한다.

2.2 기존의 데이터 이동 방법

2.1절에서 살펴 본 것처럼 Sarda는 HDBMS가 이력 릴레이션을 과거, 현재, 미래 세그먼트로 분리해서 관리해야 한다고 제안한 바 있다[16]. [16]에서는 유효 끝 시각이 현재 시각보다 작은 개체 버전은 과거 세그먼트에 저장하고, 유효 시작 시각이 현재 시각보다 작거나 같고 유효 끝 시각이 현재 시각보다 크거나 같은 개체 버전은 현재 세그먼트에 저장한다. 미래 세그먼트에는 유효 시작 시각이 현재 시각보다 큰 개체 버전을 저장한다.

[16]에서는 시간지원 데이터의 이동에 대해서 언급하고 있는데 이 연구에 의하면, 미래 세그먼트에서 현재 세그먼트로의 데이터 이동은 현재 시각이 미래 세그먼트에 들어있는 개체 버전의 유효 시각 시각과 같아지면 일어난다. 마찬가지로 현재 세그먼트에 들어있는 개체 버전 중에서 유효 끝 시각이 현재 시각과 같아지면 현재 세그먼트에서 과거 세그먼트로 데이터 이동이 발생한다.

Sarda가 언급한 데이터 이동 방식대로 한다면, 각 개체 버전이 가지고 있는 유효 시간 단위마다 각 세그먼트에 들어있는 개체 버전의 유효성을 검사하고 그 세그먼트에서 더 이상 유효하지 않은 개체 버전은 다른 세그먼트로 이동해야 한다. Sarda는 시간단위마다 세그먼트에 들어있는 개체 버전의 유효성을 검사하고 이동하는 부하가 어느 정도인지는 구체적으로 다루지 않고

개념적 수준에서 데이터 이동에 대해서 언급하였다.

앞의 2.1절에서 살펴 본 것처럼 Kouramajian은 [12]에서 시간지원 데이터의 저장 장소를 광 디스크와 자기 디스크로 구분해서 먼 과거에 유효했던 개체 버전은 광 디스크에 저장하고 가까운 과거에 유효했던 개체 버전과 현재 유효한 개체 버전은 자기 디스크에 저장한다. Kouramajian은 광 디스크과 자기 디스크에 저장될 개체 버전을 구분하는 경계값을 제시하고 있는데, 현재 유효한 개체 버전 중에서 가장 작은 유효 시작 시각을 경계값으로 하고 이 경계값을 T_m이라 한다.

T_m보다 작은 유효 끝 시각을 가지는 개체 버전은 대부분 광 디스크에 저장하고 참조한다. T_m보다 유효 시작 시각은 작고, 유효 끝 시각은 T_m보다 큰 개체 버전은 광 디스크와 자기 디스크 양쪽 모두에 저장하고 참조한다. 유효 시작 시각이 T_m보다 큰 개체 버전은 자기 디스크에 저장하고 참조한다. 광 디스크에만 저장되는 과거 개체 버전은 이력 버전(hisroty verstion)이라고 하고, 광 디스크와 자기 디스크 양쪽에 모두 저장되는 개체 버전은 걸친 버전(spanned version)이라고 하였다. 또한, 자기 디스크에만 저장하는 현재 버전을 걸치지 않은 버전(non-spanned version)이라고 구분하였다.

이 논문에서 제안하는 시간지원 데이터의 시간적 특성을 고려한 분리 저장 구조에서 다룰 구체적인 연구 내용은 다음과 같다. 먼저, 기존의 관계형 데이터베이스 시스템에 데이터 이동 프로세스(Migration Processor)와 과거, 현재, 그리고 미래 세그먼트를 추가한 분리 저장 구조를 제안하고, 분리 저장 구조에서 데이터 이동 프로세서를 구성하는 데이터 이동 검사기(Migration Detector)와 데이터 이동 실행자(Migration Executor)의 기능을 제시한다. 또한, 데이터 이동에서 옮김과 복사의 대상이 되는 개체 버전과 데이터 이동 알고리즘을 정의하고, 시간단위에 의한 즉시 이동 방법과 LST-GET와 같은 경계값에 의한 지연 이동 방법에서 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트에 저장되는 개체 버전을 정의한다.

또한, 이 논문에서는 두 가지 데이터 이동 방법을 제안한다. 먼저, 유효 시간의 시간단위마다 데이터를 이동하는 시간단위에 의한 이동 방법에서는, 각 세그먼트에 저장되어 있는 개체 버전의 유효성을 검사하는 시기와 이동 대상을 결정하는 방법을 제시한다. 그리고 Kouramajian이 제안한 데이터 이동 방법을 확장해서 미래 데이터를 처리하는 LST-GET에 의한 이동 방법에서는 시간단위에 의한 즉시 이동 방법의 문제점을 제시하고, 과거, 현재, 그리고 미래 세그먼트를 구분하는 경계값으로 사용하는 LST와 GET를 정의한다. 또한, 과거, 현재, 그리고 미래 세그먼트 가운데 한 세그먼트에만 저장되는 순수 개체 버전과 두 개 또는 세 개 세그먼트에 중복해서 저장되는 개체 버전을 정의한다.

그리고 기존의 관계형 데이터베이스 시스템에서 분리하지 않은 저장 구조와 이 논문에서 제안하는 분리 저장 구조에 대해서 질의에 대한 성능을 비교하고 분석한다. 또한, 이 논문에서 제안하는 두 가지 데이터 이동 방법에 대해서 사용자 질의에 대한 성능을 평가한다.

3. P-C-F 저장 구조와 데이터 이동 방법

이 장에서는 먼저, 논문에서 사용하는 용어와 기호에 대해서 정의하고, 3.2절에서는 제안하는 분리 저장 구조인 P-C-F 저장 구조에서 데이터 이동 프로세서의 기능을 제시한다. 3.3절에서는 P-C-F 저장 구조를 기반으로 하는 데이터의 이동 방법에 대해서 살펴보도록 한다.

3.1 용어 정의

이 논문에서 사용하는 용어와 기호의 의미는 다음과 같다.

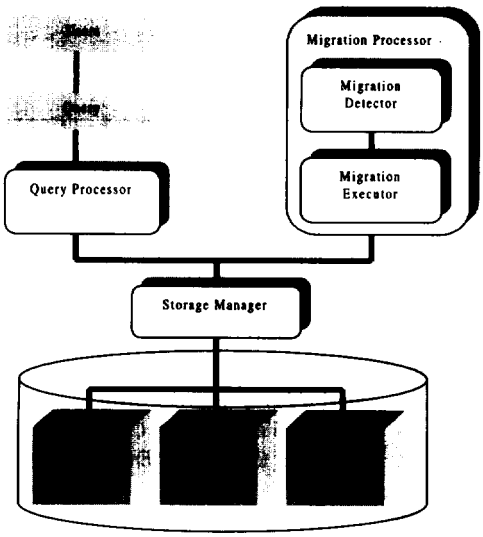
- **유효 시간(valid time)** : 어떤 사실이 실세계에서 유효한 시간[7].
- **트랜잭션 시간(transaction time)** : 어떤 사실이 데이터베이스에 기록된 시간[7].
- **시간단위(time granule)** : 최소 시간 간격의 단위, 보기 : 초, 분, 시간 등.
- **과거 데이터(past data)** : 과거 세그먼트에 저장되어 있는 데이터.
- **현재 데이터(current data)** : 현재 세그먼트에 저장되어 있는 데이터.
- **미래 데이터(future data)** : 미래 세그먼트에 저장되어 있는 데이터.
- **P-C-F 저장 구조** : 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트로 구성되는 저장 구조.
- ***PCB*(Past Current Bound)** : 과거 세그먼트와 현재 세그먼트를 구분하는 시간축(time line) 위의 시점.
- ***CFB*(Current Future Bound)** : 현재 세그먼트와

미래 세그먼트를 구분하는 시간축 위의 시점.

- ***now*** : 현재 시각.
- E_{ij} : 개체 E_i의 j번째 버전.
- $E_{ij}.A$: 개체 버전 E_{ij}의 모든 애트리뷰트.
- $E_{ij}.V_s$: 개체 버전 E_{ij}의 유효 시작 시각.
- $E_{ij}.V_e$: 개체 버전 E_{ij}의 유효 끝 시각.
- $[E_{ij}.V_s, E_{ij}.V_e]$: 닫힌 시간 간격.
- $[E_{ij}.V_s, E_{ij}.V_e)$: 열린 시간 간격, 의미는 $[E_{ij}.V_s, E_{ij}.V_e - 1]$.

3.2 P-C-F 저장 구조

1장의 서론에서 살펴 본 것처럼 시간지원 데이터의 시간적 특성을 고려해서 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트로 분리한 저장 구조에서는 시간지원 질의에 대해서 응답 속도를 향상시킬 수 있다. (그림 1)은 이 논문에서 제안하는 시간지원 데이터의 시간적 특성을 고려해서 세 개의 세그먼트로 분리한 P-C-F 저장 구조를 나타낸다.

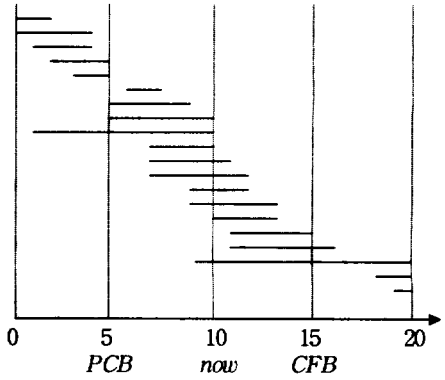


(그림 1) P-C-F 저장 구조도
(Fig. 1) P-C-F storage structure diagram

한 세그먼트에 과거, 현재, 미래 데이터를 모두 저장하는 기존의 데이터 저장 방법은 (그림 1)에서 데이터 이동 프로세서(Migration Processor)를 빼고, 과거 세그먼트(Past Segment), 현재 세그먼트(Current Segment), 그리고 미래 세그먼트(Future Segment)를 묶어서 한 세그먼트로 보면 된다. 기존의 데이터 저장 방법에서 질의를 처리하는 과정은 일반적인 관계형 데이터베이스 시스템에서 질의를 처리하는 방법과 같다.

이 논문에서 제안하는 시간지원 데이터의 시간적 특성을 고려한 P-C-F 저장 구조는 기존의 관계형 데이터베이스 시스템에 데이터 이동 프로세서가 추가로 필요하고 저장 장소로 세 개의 세그먼트가 필요하다. P-C-F저장 구조에서 데이터 이동 프로세서는 **데이터 이동 검사기**(Migration Detector)와 **데이터 이동 실행자**(Migration Executor)로 구성된다. 데이터 이동 검사기는 세그먼트 사이에 개체 버전을 이동할 시기를 결정하고, 이동 시기가 결정되면 데이터 이동 실행자를 실행시킨다. 세그먼트 사이에 개체 버전을 이동할 시기는 매 시간단위가 되거나 현재 시각이 경계값과 같아지는 순간이 될 수 있다.

이 논문에서는 데이터 이동 방법으로써 시간단위에 의한 이동 방법과 LST-GET에 의한 이동 방법을 제안하는데, 이 중에서 시간단위에 의한 이동 방법을 쓰는 경우에는 매 시간단위가 개체 버전을 이동할 시기가 된다. 그리고 LST-GET에 의한 이동 방법에서는 현재 시각이 현재 세그먼트와 미래 세그먼트를 나누는 경계값으로 쓰이는 *CFB*와 같아지는 순간이 개체 버전을 이동하는 시기가 된다. (그림 2)는 현재 시각과 경계값 사이의 관계를 나타낸 보기이다. 이 그림에서 현재 시각이 10이고 *CFB*가 15라고 가정하면, 경계값에 의한 지연 이동 방법에서는 시간이 지나서 *now*가 *CFB*인 15와 같아지면 데이터를 이동하는 시기가 된 것이다.



(그림 2) 현재 시각과 경계값 사이의 관계를 나타낸 보기
(Fig. 2) Example of relationship between now and boundary value

데이터 이동 검사기가 데이터 이동 시기를 결정하면 데이터 이동 실행자를 실행시킨다. 데이터 이동 실행자는 각 데이터 이동 방법에 따라 원래 세그먼트에서 다른 세그먼트로 개체 버전을 옮기거나 복사한다. 개체 버전을 다른 세그먼트로 옮기는 과정에는 다른 세그먼트에 쓰고 원래의 세그먼트에서 지우는 과정이 필요하다("옮김"이라고 한다). 복사하는 과정에는 다른 세그먼트에 복사하고 원래 세그먼트에도 남게 된다("복사"라고 한다).

시간단위에 의한 이동 방법에서는 매 시간단위에서 세그먼트의 유효성을 검사하고, 이동이 필요한 개체 버전은 이동하므로 옮김만 있고 복사는 없다. 경계값에 의한 지연 이동 방법인 LST-GET에 의한 이동 방법에서는 경계값을 사용하므로 복사와 옮김이 모두 필요하게 된다. 시간단위에 의한 이동 방법을 쓰는 경우에 데이터 이동 실행자는 미래 세그먼트를 검사해서 유효 시작 시각이 현재 시각보다 작고 유효 끝 시각이 현재 시각보다 크거나 같으면 현재 세그먼트로 옮기고, 현재 세그먼트를 검사해서 어떤 개체 버전의 유효 끝 시각이 현재 시각보다 작으면 과거 세그먼트로 옮긴다.

데이터 이동 실행자는 *PCB*와 *CFB*를 기준으로 시간지원 데이터를 미래 세그먼트에서 현재 세그먼트로 옮기거나 복사할 개체 버전을 결정하고, 현재 세그먼트에서 과거 세그먼트로 옮기거나 복사할 개체 버전을 결정한다. (그림 3)은 각 세그먼트에서 옮김이나 복사의 대상이 되는 개체 버전을 나타낸 것이다.

데이터의 옮김, 복사, 지우기	해당 개체 버전
미래 세그먼트 → 현재 세그먼트로 옮김	$(E_{ij}.V_s \geq PCB \wedge E_{ij}.V_e < CFB)$
현재 세그먼트 → 과거 세그먼트로 옮김	$(E_{ij}.V_e < PCB)$
미래 세그먼트 → 현재 세그먼트로 복사	$(PCB \leq E_{ij}.V_s < CFB) \wedge (E_{ij}.V_e > CFB)$
현재 세그먼트 → 과거 세그먼트로 복사	$(E_{ij}.V_s < PCB) \wedge (PCB < E_{ij}.V_e < CFB)$

(그림 3) 옮김과 복사의 대상이 되는 개체 버전
(Fig. 3) Entity versions the object of moving and copy

(그림 5)는 P-C-F 저장 구조에서 데이터 이동 실행자가 데이터를 이동하는 과정을 나타낸 것이다. 이 데이터 이동 알고리즘에서 사용하는 함수와 변수는 (그림 4)와 같다. 시간단위에 의한 이동 방법에서는 *PCB* = *now*가 되고, *CFB* = *now* + 1이 된다. 경계값에 의한 지연 이동 방법에서 *PCB*와 *CFB*를 결정하는 방법은 4장에서 구체적으로 살펴보도록 한다.

read(*Segment*) : 미래 세그먼트 또는 현재 세그먼트에서 한 개체 버전을 읽는 함수.
WriteToCur(*Buffer*) : 임시 기억 장소로 사용하는 버퍼에 들어있는 내용을 현재 세그먼트에 쓰는 함수.
WriteToFut(*Buffer*) : 임시 기억 장소로 사용하는 버퍼에 들어있는 내용을 미래 세그먼트에 쓰는 함수.
append(*Buffer*, *EntityVersion*) : 개체 버전을 버퍼에 추가하는 함수.
EntityVersion : 한 개체 버전을 저장하는 변수.
EntityVersion.Vs : 한 개체 버전의 시작 유효 시각을 나타내는 변수.
FutSeg : 미래 세그먼트를 나타내는 변수.
CurSeg : 현재 세그먼트를 나타내는 변수.
MigBuf : 다른 세그먼트로 옮겨야 할 개체 버전을 저장하는 버퍼를 나타내는 변수.
StayFurBuf : 미래 세그먼트에 쓸 개체 버전을 저장하는 버퍼를 나타내는 변수.
StayCurBuf : 현재 세그먼트에 쓸 개체 버전을 저장하는 버퍼를 나타내는 변수.

(그림 4) 데이터 이동 알고리즘에 쓰는 함수와 변수
(Fig. 4) Functions and variables used in data migration algorithm

```
DataMigration()
begin
  while (not eof (EntityVersion <- read (FutSeg)) do
    if  EntityVersion.Vs < CFB then
        append (MigBuf, EntityVersion);
        if  EntityVersion.Ve >= CFB then
            append (StayFutBuf, EntityVersion);
        end if
    else
        append (StayFutBuf, EntityVersion);
    end if
    if (MigBuf is full) then
        WriteToCur (MigBuf);
    end if
    if (StayFutBuf is full) then
        WriteToFut (StayFutBuf);
    end if
  end while
  MigBuf <- Null;
```

```
while (not eof (EntityVersion <- read (CurSeg)) do
  if EntityVersion.Ve < PCB then
    append (MigBuf, EntityVersion);
      if EntityVersion.Ve >= PCB then
        append (StayCurBuf, EntityVersion);
      end if
  else
    append (StayCurBuf, EntityVersion);
  end if
  if (MigBuf is full) then
    WriteToPast (MigBuf);
  end if
  if (StayCurBuf is full) then
    WriteToCur (StayCurBuf);
  end if
 end while
end
```

(그림 5) 데이터 이동 알고리즘
(Fig. 5) Data migration algorithm

3.3 시간지원 데이터의 이동 방법

이 절에서는 P-C-F 저장 구조를 기반으로 하는 데이터의 이동방법에 대해서 살펴보도록 한다. 먼저, 3.3.1절에서 시간단위에 의한 이동 방법을 살펴보고, 3.3.2절에서는 LST-GET에 의한 이동 방법을 살펴본다.

3.3.1 시간단위에 의한 이동 방법

시간지원 데이터의 유효 시간과 현재 시각의 관계에 따라 과거에 유효했던 개체 버전, 현재 유효한 개체 버전, 미래에 유효한 개체 버전으로 나눌 수 있다. 시간단위 이동 방법에서는 <표 1>에서 보인 것처럼, 현재 시각을 기준으로 과거 데이터, 현재 데이터, 그리고 미래 데이터로 구분한다.

〈표 1〉 시간단위에 의한 이동 방법에서 시간지원 데이터를 구분하는 조건
〈Table 1〉 Conditions distinguish temporal data in the migration method by time granularity

$E_{ij}.V_s$ \ $E_{ij}.V_e$	$E_{ij}.V_e$<now	$E_{ij}.V_e$=now	$E_{ij}.V_e$>now
$E_{ij}.V_s$ < now	과거 데이터	과거 데이터	현재 데이터
$E_{ij}.V_s$ = now	×	×	현재 데이터
$E_{ij}.V_s$ > now	×	×	미래 데이터

즉, 현재 시각을 기준으로 어떤 개체 버전의 유효 끝 시각이 현재 시각보다 작으면 과거 세그먼트에 저장하고, 유효 시작 시각이 현재 시각보다 작거나 같고 유효 끝 시각이 현재 시각보다 크면 현재 세그먼트에 저장한다. 그리고 유효 시작 시각이 현재 시각보다 큰 개체 버전은 미래 세그먼트에 저장한다. 이 관계는 다음과 같이 정의할 수 있다.

- 과거 세그먼트에 들어가는 개체 버전의 집합
  $= \{ E_{ij} \mid E_{ij}.V_e \leq now \}$
- 현재 세그먼트에 들어가는 개체 버전의 집합
  $= \{ E_{ij} \mid E_{ij}.V_s \leq now < E_{ij}.V_e \}$
- 미래 세그먼트에 들어가는 개체 버전의 집합
  $= \{ E_{ij} \mid E_{ij}.V_s > now \}$

과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트에 들어있는 개체 버전의 집합에서 각 세그먼트의 개체 버전이 가지고 있는 유효 시간 간격의 집합을 각각 I_p, I_c, 그리고 I_f라고 하고 다음과 같이 정의한다.

- I_p = 과거 세그먼트에 들어있는 모든 개체 버전의 유효 시간 간격의 집합
  $= \{ i_{p1}, i_{p2}, \cdots, i_{pl} \}$
- I_c = 현재 세그먼트에 들어있는 모든 개체 버전의 유효 시간 간격의 집합
  $= \{ i_{c1}, i_{c2}, \cdots, i_{cm} \}$
- I_f = 미래 세그먼트에 들어있는 모든 개체 버전의 유효 시간 간격의 집합
  $= \{ i_{f1}, i_{f2}, \cdots, i_{fn} \}$

각 세그먼트에 들어있는 개체 버전의 유효 시간 간격에서 가장 작은 유효 시작 시각을 찾는 함수를 *MinStart*(), 가장 큰 유효 끝 시각을 찾는 함수를 *MaxEnd*()라고 하면, I_p, I_c, 그리고 I_f 각각에서 최소값과 최대값을 구할 수 있다.

I_p의 최소값을 $MinStart_{ip}$, 최대값을 $MaxEnd_{ip}$라고 하고, I_c의 최소값을 $MinStart_{ic}$, 최대값을 $MaxEnd_{ic}$라고 한다. 그리고 I_f의 최소값을 $MinStart_{if}$, 최대값을 $MaxEnd_{if}$라고 한다. I_p, I_c, 그리고 I_f의 lifespan을 각각 $L(I_p)$, $L(I_c)$, 그리고 $L(I_f)$라고 하고 다음과 같이 정의한다.

$$L(I_p) = [MinStart_{ip}, MaxEnd_{ip})$$
$$L(I_c) = [MinStart_{ic}, MaxEnd_{ic})$$
$$L(I_f) = [MinStart_{if}, MaxEnd_{if})$$

데이터 이동 프로세서가 현재 세그먼트를 검사하는 범위는 $MinStart_{ic}$에서 $MaxEnd_{ic}$까지 이고, 미래 세그먼트를 검사하는 범위는 $MinStart_{if}$에서 $MaxEnd_{if}$까지가 된다. 데이터 이동 프로세서는 현재 세그먼트와 미래 세그먼트를 차례로 검사하게 되므로 그 범위가 $MinStart_{ic}$에서 $MaxEnd_{if}$까지가 된다. 이것을 *lifespan*으로 나타내면 다음과 같다.

$$L(L(I_c) \cup L(I_f)) = [MinStart_{ic}, MaxEnd_{if})$$

과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트의 시간 범위는 각각 $L(I_p)$, $L(I_c)$, 그리고 $L(I_f)$이고, 이것은 시점 질의와 범위 질의가 들어왔을 때 검색 대상이 되는 세그먼트를 정하는 기준이 된다. 또한, $L(I_c)$와 $L(I_f)$는 데이터의 유효성 검사의 대상이 되는 현재 세그먼트와 미래 세그먼트의 시간 범위가 된다.

시간이 흐름에 따라서 미래는 현재가 되고 현재는 과거가 되기 때문에, P-C-F 저장 구조에서는 미래 세그먼트에 저장되어 있는 개체 버전 중에서 유효 시작 시각이 현재 시각보다 작거나 같고, 유효 끝 시각이 현재 시각보다 크면 현재 세그먼트로 옮겨야 한다. 또한, 현재 세그먼트에 저장되어 있는 개체 버전 중에서 유효 끝 시각이 현재 시각보다 작거나 같으면 과거 세그먼트로 옮겨야 한다.

따라서 미래 세그먼트나 현재 세그먼트를 검사해서 이동해야 할 개체 버전은 다른 세그먼트로 옮김으로써 시간적 유효성을 보장할 수 있다. 미래 세그먼트와 현재 세그먼트에서 이동할 개체 버전을 찾기 위해서는 적절한 시기에 세그먼트에 들어있는 개체 버전들을 검사해야 되는데, 각 세그먼트를 검사하는 시점을 결정하는 방법에는 다음과 같이 세 가지가 있다.

1) 시간지원 데이터의 시간단위(time granule)마다 미래 세그먼트와 현재 세그먼트를 주기적으로 검사한다.
2) 현재 시각이 현재와 미래를 나누는 경계값(CFB)과 같으면 미래 세그먼트를 검사한다.
3) 질의가 처리될 때마다 미래 세그먼트와 현재 세그먼트를 검사한다.

질의가 처리될 때마다 미래 세그먼트와 현재 세그먼트에 저장되어 있는 개체 버전이 시간적 유효성을 가지고 있는지 검사하는 세 번째 방법은 시스템에 주는 부담이 너무 크기 때문에 이 방법은 적합하지 않다.

첫 번째 방법에서는 시간지원 데이터의 시간단위마다 미래 세그먼트와 현재 세그먼트에 저장되어 있는 개체 버전을 검사해서 그 세그먼트에서 더 이상 유효하지 않은 데이터는 다른 세그먼트로 옮겨야 하는데, 이 때 기준이 되는 시간단위는 크기에 따라서 일반적으로 초, 분, 시간, 일, 주, 월, 년 중에 하나가 될 수 있다. 이러한 데이터 이동의 기준이 되는 시간단위는 시간지원 데이터의 시간단위가 되는데, 시간단위에 의한 이동 방법에서는 매 시간단위가 세그먼트를 검사하는 시기가 된다. 두 번째 방법은 3.3.2절에서 자세히 살펴보도록 한다.

하나의 시간지원 데이터베이스 안에서도 각 릴레이션의 시간지원 데이터가 서로 다른 시간단위를 가질 수 있다. 이런 경우는 시간단위를 변환하는 추가의 연산이 필요하다[3]. P-C-F 저장 구조를 가지는 시간지원 데이터베이스에서 시간단위에 의한 이동 방법을 쓰는 경우에 각 세그먼트에 저장되어 있는 개체 버전들의 유효성을 유지하기 위해서 시간단위마다 주기적으로 데이터 이동이 필요하다. 시간단위에 의한 이동 방법에서 데이터를 이동하는 알고리즘은 (그림 5)에서 살펴 본 바와 같이 $PCB = now$이고 $CFB = now + 1$로 두면 된다.

시간단위에 의한 이동 방법에서 검사 대상이 되는 개체 버전의 집합은 현재 세그먼트에 들어있는 개체 버전의 집합과 미래 세그먼트에 들어있는 개체 버전 집합이 된다. 시간단위에 의한 이동 방법에서는 한 릴레이션에서 유효 시간의 시간단위가 초이면 매 초마다 데이터 이동이 발생하고, 시간단위가 분이면 매 분마다 데이터 이동이 발생한다.

3.3.2 LST-GET에 의한 이동 방법

2장에서 살펴 본 것처럼 [12]에서는 시간지원 데이터를 저장하기 위해서 두 개의 기억 장소인 광 디스크와 자기 디스크를 이용하였다. 현재 유효한 개체 버전 중에서 유효 시작 시각이 가장 작은 개체 버전의 유효 시작 시각을 기준으로 해서 이 시각보다 작으면 광 디스크에 저장하고 이 시각보다 크면 자기 디스크에 저장하였다.

[12]에서는 미래 데이터를 고려하지 않았는데, 이 논문에서는 [12]에서 두 저장 매체에 저장하는 데이터를 구분하는 경계값으로 사용한 최소 유효 시작 시각의 개념을 확장해서 미래 데이터에 적용할 수 있는 *GET*를

제안한다. 현재 시각에 유효한 개체 버전의 유효 시간 집합에서 유효 시작 시각이 가장 작은 개체 버전의 유효 시작 시각을 *LST*라고 하고 다음과 같이 정의한다.

$$LST = \min \{ E_{ij}.V_s \mid E_{ij}.V_s \le now < E_{ij}.V_e \}$$

현재 시각에 유효한 개체 버전의 유효 시간 집합에서 유효 끝 시각이 가장 큰 개체 버전의 유효 끝 시각을 *GET*라고 하고, 다음과 같이 정의한다.

$$GET = \max \{ E_{ij}.V_e \mid E_{ij}.V_s \le now < E_{ij}.V_e \}$$

*LST*와 *GET*를 기준으로 시간지원 데이터를 과거 데이터, 현재 데이터, 그리고 미래 데이터로 나누어서 저장한다. $E_{ij}.V_e \le LST$를 만족하는 시간지원 데이터를 과거 데이터라 하고 이 데이터는 과거 세그먼트에 저장한다. $(E_{ij}.V_s \ge LST) \wedge (E_{ij}.V_e < GET)$를 만족하는 시간지원 데이터를 현재 데이터라 하고 현재 세그먼트에 저장한다. $E_{ij}.V_s \ge GET$를 만족하는 시간지원 데이터를 미래 데이터라 하고 미래 세그먼트에 저장한다.

$(E_{ij}.V_s < LST) \wedge (LST < E_{ij}.V_e < GET)$를 만족하는 시간지원 데이터는 과거 세그먼트와 현재 세그먼트에 중복해서 저장한다. $(LST \le E_{ij}.V_s < GET) \wedge (E_{ij}.V_e > GET)$를 만족하는 시간지원 데이터는 현재 세그먼트와 미래 세그먼트에 중복해서 저장한다. 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트에 들어가는 개체 버전의 집합을 정의하면 다음과 같다.

- 과거 세그먼트에 들어가는 순수 개체 버전의 집합 $= \{ E_{ij} \mid E_{ij}.V_e < LST \} = P$
- 현재 세그먼트에 들어가는 순수 개체 버전의 집합 $= \{ E_{ij} \mid (E_{ij}.V_s \ge LST) \wedge (E_{ij}.V_e < GET) \} = C$
- 미래 세그먼트에 들어가는 순수 개체 버전의 집합 $= \{ E_{ij} \mid E_{ij}.V_s \ge GET \} = F$

과거 세그먼트에 들어가는 순수 개체 버전의 집합은 과거 세그먼트와 현재 세그먼트에 중복해서 들어가지 않고 오로지 과거 세그먼트에만 들어가는 개체 버전을 말한다. 현재 세그먼트와 미래 세그먼트에서도 같은 의미로 사용한다. 과거 세그먼트와 현재 세그먼트에 중복해서 저장하고, 현재 세그먼트와 미래 세그먼트에 중복해서 저장하는 개체 버전은 다음과 같이 정의한다.

- 과거 세그먼트와 현재 세그먼트에 중복해서 들어가는 개체 버전의 집합 $= \{E_{ij} \mid E_{ij}.V_s < LST \wedge LST < E_{ij}.V_e < GET\} = PC$
- 현재 세그먼트와 미래 세그먼트에 중복해서 들어가는 개체 버전의 집합 $= \{E_{ij} \mid (LST \le E_{ij}.Vs < GET) \wedge (E_{ij}.Ve > GET)\} = CF$

경계값 *LST*와 *GET*에 의해서 각 세그먼트에 저장되는 개체 버전은 (그림 6)과 같다.

경계값과 유효 시간의 관계	저장되는 세그먼트
$E_{ij}.V_e < LST$	과거 세그먼트
$(E_{ij}.V_s \ge LST) \wedge (LST < E_{ij}.V_e < GET)$	현재 세그먼트
$E_{ij}.V_s \ge GET$	미래 세그먼트
$(E_{ij}.V_s < LST) \wedge (LST < E_{ij}.V_e < GET)$	과거, 현재 세그먼트
$(LST \le E_{ij}.V_s < GET) \wedge (E_{ij}.V_e > GET)$	현재, 미래 세그먼트

(그림 6) LST와 GET에 의한 개체 버전의 저장 세그먼트
(Fig. 6) Storage segments of entity versions by LST and GET

현재 시각이 *GET*를 만나면 새로운 *LST*와 *GET*를 구하는데, 새로운 *LST*는 현재 시각에 유효한 개체 버전 중에서 유효 시작 시각이 가장 작은 개체 버전의 유효 시작 시각이 된다. 또한, 새로운 *GET*는 현재 시각에 유효한 개체 버전 중에서 유효 끝 시각이 가장 큰 개체 버전의 유효 끝 시각이 된다. 새로운 *LST*와 *GET*는 직전의 *LST*와 *GET*보다 작은 값은 선택하지 않는다.

새로운 *LST*를 구하는데 해당하는 개체 버전의 집합은, 집합 *C*에 들어가는 개체 버전 중에서 유효 시작 시각이 *LST*보다 큰 개체 버전의 집합이 되고 이 집합을 V_C 라고 하면 다음과 같이 정의할 수 있다.

$$V_C = \{E_{ij} \mid (E_{ij}.V_s < now) \wedge (E_{ij}.V_e > now) \wedge (E_{ij}.V_s > LST)\}$$

V_C에 들어가는 개체 버전의 유효 시간 간격의 집합을 IV_C라고 하고 이것을 다음과 같이 나타낸다.

$$IV_C = \{ v_{c1}, v_{c2}, \cdots, v_{cn} \}$$

IV_C의 최소값을 *Tmin_vc*, 최대값을 *Tmax_vc*라고 하면 새로운 *LST*는 *Tmin_vc*가 된다.

새로운 *GET*를 구하는데 해당하는 개체 버전의 집합은, 집합 *F*에 들어가는 개체 버전 중에서 유효 끝 시각이 *GET*보다 큰 개체 버전의 집합이 되고 이 집합을 V_F라고 하면 다음과 같이 정의할 수 있다.

$$V_F = \{E_{ij} \mid (E_{ij}.V_s < now) \wedge (E_{ij}.V_e \geq now) \wedge (E_{ij}.V_e > GET)\}$$

집합 V_F에 포함되는 개체 버전의 유효 시간 간격의 집합을 IV_F라고 하고, 이것을 다음과 같이 나타낸다.

$$IV_F = \{ v_{f1}, v_{f2}, \cdots, v_{fn} \}$$

집합 IV_F에서 최소값과 최대값을 각각 *Tmin_uf*, *Tmax_uf*라고 하면 *Tmax_uf*가 새로운 *GET*가 된다.

*LST*를 구하는 과정을 살펴 보면 다음과 같다. 먼저, 현재 시각에 유효한 개체 버전 중에서 유효 시작 시각이 가장 작고, 현재의 *LST*보다 큰 유효 시작 시각을 가지는 개체 버전을 찾는다. 이렇게 찾은 개체의 유효 시작 시각을 *LST*로 한다. *LST*는 과거 세그먼트와 현재 세그먼트에 저장될 개체 버전을 구분하는 데 사용한다. *LST*를 구하는 과정을 알고리즘으로 나타내면 (그림 7)과 같다.

```
GetLST()
begin
 /* for all E_ij in Future Segment and Current Segment */
   LSTpre <- LST;  LST <- Null;
   while (not eof (EntityVersion <- read (FutSeg))) do
    if ( EnitityVersion.V_s ≤ now < EntityVersion.V_e )
          ∧ ( EntityVersion.V_s > LSTpre ) then
       CompVersion.V_s <- EntityVersion.V_s;
       LST <- min ( CompVersion.V_s, LST);
    end if
 end while
   while (not eof (EntityVersion <- read (CurSeg))) do
    if ( EnitityVersion.V_s ≤ now < EntityVersion.V_e )
          ∧ ( EntityVersion.V_s > LSTpre ) then
       CompVersion.V_s <- EntityVersion.V_s;
       LST <- min ( CompVersion.V_s, LST);
    end if
 end while
 if (LST = Null) then
     LST <- Notfound;
 end if
 return LST;
end
```

(그림 7) *LST*를 구하는 알고리즘
(Fig. 7) Getting LST algorithm

```
GetGET()
begin
/* for all E_ij in Future Segment and Current Segment */
  GETpre <- GET;  GET <- Null;
  while (not eof (EntityVersion <- read (FutSeg))) do
   if ( EnitityVersion.V_s ≤ now < EntityVersion.V_e )
         ∧ ( EntityVersion.V_e > GETpre ) then
      CompVersion.V_e <- EntityVersion.V_e;
      GET <- max ( CompVersion.V_e, GET);
   end if
end while
  while (not eof (EntityVersion <- read (CurSeg))) do
   if ( EnitityVersion.V_s ≤ now < EntityVersion.V_e )
         ∧ ( EntityVersion.V_e > GETpre ) then
      CompVersion.V_e <- EntityVersion.V_e;
      GET <- max ( CompVersion.V_e, GET);
   end if
 end while
 if (GET = Null) then
     GET <- Notfound;
 end if
 return GET;
end
```

(그림 8) *GET*를 구하는 알고리즘
(Fig. 8) Getting GET algorithm

*GET*를 구하는 과정을 살펴보면 먼저, 현재 시각에 유효한 개체 버전 중에서 유효 끝 시각이 가장 크고, 현재의 *GET*보다 큰 유효 끝 시각을 가지는 개체 버전의 유효 끝 시각을 새로운 *GET*로 한다[그림 8]. *GET*는 현재 세그먼트와 미래 세그먼트에 저장될 개체 버전을 결정하는 기준으로 사용한다. 데이터 이동은 옮김과 복사로 나누어진다. *LST* 또는 *GET*에 걸치는 개체 버전은 경계값을 중심으로 양쪽 세그먼트에 중복해서 저장해 두는 복사를 하게 된다. 각 세그먼트 사이에 데이터를 옮기고 복사하는 경우는 다음과 같이 분류할 수 있다[(그림 9, 10)].

- 미래 세그먼트에서 현재 세그먼트로 옮김
- 현재 세그먼트에서 과거 세그먼트로 옮김
- 미래 세그먼트에서 현재 세그먼트로 복사
- 현재 세그먼트에서 과거 세그먼트로 복사

개체 버전을 복사한 뒤에 지우기 과정이 필요한 옮김은 미래 세그먼트에 현재 세그먼트로 옮김과 현재 세그먼트에서 과거 세그먼트로 옮김이 있다. 새로운

*LST*와 *GET*를 적용하기 직전에는 유효 시작 시각이 직전의 *GET*보다 크기 때문에 미래 세그먼트에 저장되어 있다가 다음 상태에서는 유효 시간의 시작이 *LST*보다 크거나 같고 유효 끝 시각이 *GET*보다 작은 개체 버전은 현재 세그먼트에 들어가야 한다.

따라서, 현재 세그먼트에 복사하고 미래 세그먼트에서는 지우게 된다. 또한, 직전의 *LST*와 직전의 *GET*사이에 포함되어서 현재 세그먼트에 저장되어 있던 개체 버전이 그 유효 끝 시각이 새로 구한 *GET*보다 작은 경우에는 과거 세그먼트에 저장되어야 한다. 이러한 개체 버전은 과거 세그먼트에 복사하고 현재 세그먼트에서 지운다.

경계값을 중심으로 개체 버전이 양쪽 세그먼트에 중복해서 존재하도록 하는 복사는 미래 세그먼트에서 현재 세그먼트로 복사하거나 현재 세그먼트에서 과거 세그먼트로 복사한다. 새로운 *LST*와 *GET*를 적용하기 직전에는 개체 버전의 유효 시작 시각이 직전의 *GET*보다 크기 때문에 미래 세그먼트에 들어 있다가, 새로운 *LST*와 *GET*에 의하여 유효 시작 시각이 *LST*보다 크거나 같고 *GET*보다 작고, 유효 끝 시각이 *GET*보다 큰 개체 버전은 미래 세그먼트에서 현재 세그먼트에 복사된다.

새로운 경계값을 적용하기 직전 상태에서 현재 세그먼트와 미래 세그먼트에 중복해서 존재하는 개체 버전 가운데 미래 세그먼트에서 지워져야 하는 것도 있고, 현재 세그먼트와 과거 세그먼트에 중복해서 존재하는 개체 버전 가운데 현재 세그먼트에서 지워져야 하는 개체 버전이 있다.

데이터의 이동 방향	해당 개체 버전
미래 세그먼트 → 현재 세그먼트	$(E_{ij}.V_s \geq LST) \wedge (E_{ij}.V_e < GET)$
현재 세그먼트 → 과거 세그먼트	$E_{ij}.V_e < LST$

(그림 9) 옮김의 대상이 되는 개체 버전
(Fig. 9) Entity versions the object of moving

데이터의 이동 방향	해당 개체 버전
미래 세그먼트 → 현재 세그먼트	$(LST \leq E_{ij}.V_s < GET) \wedge (E_{ij}.V_e > GET)$
현재 세그먼트 → 과거 세그먼트	$(E_{ij}.Vs < LST) \wedge (E_{ij}.V_e < GET)$

(그림 10) 복사의 대상이 되는 개체 버전
(Fig. 10) Entity versions the object of copy

미래 세그먼트에서 현재 세그먼트로 데이터를 이동하는 과정과 현재 세그먼트에서 과거 세그먼트로 데이터를 이동하는 과정은 (그림 5)와 같다. LST-GET에 의한 이동 방법에서 *LST*와 *GET*가, (그림 5)에서는 각각 *PCB, CFB*에 해당한다.

4. 모의 실험

이 장에서는 분리 저장 구조와 분리하지 않은 기존의 저장 구조에 대해서 질의에 대한 성능을 분석하고, 두 가지 데이터 이동 방법에 대해서 성능을 분석한다. 먼저, 4.1절에서 모의 실험을 위한 시스템 모델을 살펴보고, 4.2절에서는 모의 실험 방법을 보고, 4.3절에서 모의 실험에 대한 결과를 분석하도록 한다.

4.1 모의 실험 모델

두 가지 데이터 이동 방법에 대해서 사용자 질의에 대한 평균 응답 시간을 측정하기 위한 시뮬레이션 모델은, (그림 1)에서 Users와 Query를 빼고 질의 생성기(Query Generator)와 질의 대기 큐(Query Waiting Queue)를 차례대로 넣은 것과 같이 설정한다. 한 세그먼트에 과거, 현재, 미래 데이터를 모두 저장하는 기존의 데이터 저장 방법은 (그림 1)에서 데이터 이동 프로세서를 빼고, 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트를 묶어서 한 개의 세그먼트로 보면 된다. 기존의 데이터 저장 방법에서 질의를 처리하는 과정은 일반적인 관계형 데이터베이스 시스템에서 질의를 처리하는 방법과 같다.

모의 실험 모델에서 질의 생성기는 시점 질의와 범위 질의를 지수 분포에 의해서 만들어내고 만들어진 질의는 질의 대기 큐에 들어간다. 질의 처리기(Query Processor)는 질의 대기 큐에서 하나의 질의를 꺼내어 시점 질의인가, 범위 질의인가를 구분하고 검색할 세그먼트를 결정한 뒤에 해당 세그먼트를 읽는다.

데이터 이동 프로세서(Migration Processor)는 데이터 이동 방법에 따라 현재 세그먼트와 미래 세그먼트를 검사해서 다른 세그먼트로 옮기거나 복사한다. 세그먼트 사이에 개체 버전을 이동하는 과정은 각각의 데이터 이동 방법에서 제시한 이동 알고리즘을 따른다. 경계값을 구하는 부하는 모의 실험에서 고려하지 않는다. 데이터를 이동하기 위해서 미래 세그먼트와 현재 세그먼트를 읽었을 때 경계값을 계산하므로 경계값을

구하기 위해서 추가로 요구되는 디스크 오버헤드는 없다고 가정한다. 경계값을 구하기 위해서 메인 메모리 상에서 일어나는 연산 오버헤드는 디스크 오버헤드에 비교하면 무시할 수 있을 정도로 작으므로 경계값을 구하는 오버헤드는 본 모의 실험에서 고려하지 않는다.

데이터 이동 프로세서가 현재 세그먼트와 미래 세그먼트를 검사하고 이동하는 중에 발생한 모든 질의는 데이터 이동이 완료될 때까지 질의 대기 큐에서 기다리고 데이터 이동이 끝나면 질의를 계속 처리한다. 분리 저장 구조에서는 데이터 이동 방법별로 데이터 이동 오버헤드를 고려해서 질의를 처리하는 시간을 측정하는 것이 주요 관점이므로, 이 실험에서는 로킹(locking)에 따른 오버헤드는 고려하지 않는다.

4.2 모의 실험 방법

모의 실험에 사용할 데이터는 TimeIT(the Time-Integrated Testbed)를 이용해서 만들었다. TimeIT는 미국 애리조나 대학의 TimeCenter에서 개발한 시간지원 질의의 연산 알고리즘을 테스트하기 위한 시험도구이다.

모의 실험에 사용할 데이터는 다음과 같다. 일반적인 데이터의 유효 시간의 길이는 30에서 50 사이에 단위가 1인 크기로 만들었고, 각각의 길이는 균등 분포(uniform distribution)로 생성하였다. LLT(Long Lived Tuples)는 일반적인 튜플의 10배 정도 큰 것으로 가정하고 LLT의 길이는 300에서 500 사이의 길이를 가지는 것으로 하고, 길이의 분포는 균등 분포로 만들었다. LLT의 비율은 0%, 1%, 3%, 5%, 7%, 그리고 9%로 각각 생성하였으며, LLT는 상대적으로 유효 시간이 긴 데이터를 말하는 것이므로 LLT의 비율이 10%가 넘는 것은 실험에서 고려하지 않는다. <표 2>에서 모의 실험에 사용할 데이터를 보인다. 모의 실험에서는 시게이트(Seagate) 사에서 나온 최근 제품인 3.5 인치 ST34501N의 하드 디스크 성능 인수값을 사용하였다 [<표 2>].

〈표 2〉 모의 실험에 사용되는 데이터
〈Table 2〉 Data used in simulation

설 명	값
일반적인 데이터의 유효 시간 길이	30, 31, … , 50
LLT의 유효 시간 길이	300, 310, …, 500
전체 시간 범위	0 ~ 10,000
LLT의 비율	0, 1, 3, 5, 7, 9%
데이터의 개수	100,000개

〈표 3〉 모의 실험에 사용한 하드 디스크의 성능 매개변수와 값
〈Table 3〉 Parameters and values of hard disk used in simulation

설 명	값
평균 탐색 시간, 읽기/쓰기 (Average Seek, Read/Write (msec))	7.7 / 8.7
트랜간 탐색 시간, 읽기/쓰기 (Track-To-Track Seek, Read/Write(msec))	0.98 / 1.24
평균 지연 시간(Average Latency(msec))	2.99
데이터 전송율(Transfer Rate(Bytes/sec))	16,777,216
트랙 크기(Track Size(Bytes))	115,078

사용자 질의의 종류는 시점 질의와 범위 질의로 구분하였으며, 모의 실험에서는 다음과 같이 세 가지 경우로 구분해서 실험하였다. 시점 질의는 과거, 현재, 그리고 미래 세그먼트 중에서 한 세그먼트만 검색 대상이 되고, 범위 질의는 한 개, 두 개, 또는 세 개의 세그먼트가 검색 대상이 될 수 있다. 시간지원 질의는 10%, 20%, 30%, 40%, 그리고 50%로 나누고 각 시간지원 질의의 비율 내에서 과거, 현재, 미래, 과거와 현재, 현재와 미래, 과거와 현재와 미래 세그먼트 각각에 균등하게(1/6씩) 시간지원 질의가 주어지도록 하였다.

모의 실험에서 적용한 가정은 다음과 같다. 첫째, 한 블록은 한 릴레이션의 튜플만을 포함한다. 둘째, 릴레이션은 비시간지원 키(non-temporal key)로 인덱스되어 있으므로 시간지원 질의인 사용자 질의는 릴레이션의 모든 튜플을 검색한다. 셋째, 사용자 질의의 평균 응답 시간에는 데이터 이동시의 지연과 질의 처리의 대상이 되는 릴레이션의 크기만을 고려한다.

4.3 모의 실험 결과 분석

이 절에서는 먼저, 시간지원 데이터를 분리하지 않고 한 세그먼트에 저장하는 기존의 저장 구조와, 시간지원 데이터의 특성을 고려한 분리 저장 구조에서 시간단위에 의한 이동 방법을 쓰는 경우에 사용자 질의에 대한 응답 시간을 측정해서 성능을 비교한다. 그리고, 시간단위에 의한 이동 방법과 LST-GET에 의한 이동 방법에 대해서 질의에 대한 성능을 분석한다.

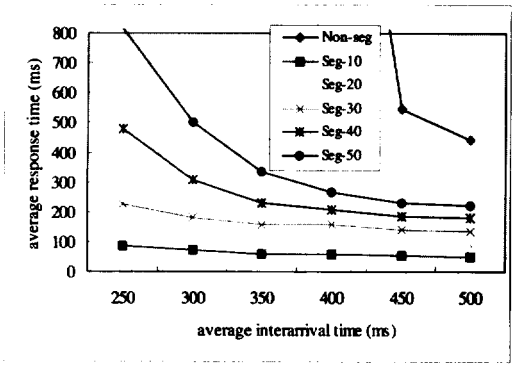
4.3.1 분리 저장 구조와 기존 저장 구조의 비교

(그림 11)과 (그림 12)는 시간단위에 의한 이동 방

법을 쓰는 경우에 분리 저장 구조와 분리하지 않은 저장 구조에 대해서 사용자 질의에 대한 평균 응답 시간을 나타낸 것이다. (그림 11)과 (그림 12)에서 Non-seg는 분리하지 않은 기존의 저장 구조를 나타내고, Seg는 분리 저장 구조를 나타낸다. (그림 11)에서 Seg-10은 시간지원 질의가10% 임을 나타내는데, 이것은 분리 저장 구조에서는 1초마다 데이터 이동이 일어나고 10%의 시간지원 질의가 과거, 현재, 미래, 과거와 현재, 현재와 미래, 과거와 현재와 미래 세그먼트에 꼭 같이 나뉘어져 검색하는 것을 뜻한다.

(그림 10)에서 분리 저장 구조에서 1초마다 데이터 이동이 일어나고 시간지원 질의가 10%인 경우에 사용자 질의의 평균 도착 시간이 450ms 이면, 사용자 질의에 대한 응답 시간이 분리하지 않은 저장 구조에서 시간지원 질의가 10%인 경우에 응답 시간보다 약 10%로 줄어들었다. 분리하지 않은 저장 구조에서 질의의 평균 도착 시간이 450ms 보다 작으면 분리하지 않은 저장 구조에서는 질의에 대한 성능이 급격히 저하된다.

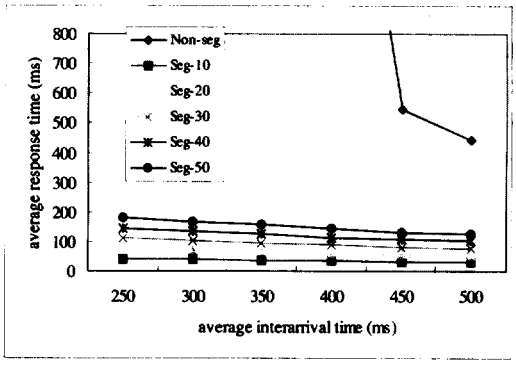
분리하지 않은 저장 구조에서는 질의가 비시간지원 질의든지 시간지원 질의든지 상관없이 과거, 현재, 그리고 미래 데이터가 모두 저장되어 있는 한 세그먼트를 검색한다. (그림 11)의 분리 저장 구조에서 시간지원 질의가 10%인 경우에는 90%를 차지하는 비시간 질의가 분리된 현재 세그먼트를 검색하고 10%의 시간지원 질의는 과거, 현재, 미래, 과거와 현재, 현재와 미래, 과거와 현재와 미래 세그먼트에 나뉘어져 검색하기 때문에, 한 세그먼트를 모두 검색하는 분리하지 않은 저장 구조보다 질의에 대한 평균 응답 시간이 줄어든다.



(그림 11) 질의에 대한 평균 응답 시간(이동 시간단위 : 초)
(Fig. 11) Average response time at second granularity level for queries

(그림 11)의 분리 저장 구조에 대해서 시간지원 질의 비율마다 사용자 질의에 대한 평균 응답 시간의 변화를 살펴보면, 시간지원 질의가 10% 에서 50%로 증가하고 사용자 질의의 평균 도착 시간이 작아질수록 질의에 대한 성능이 떨어지는 것을 알 수 있다. 시간지원 질의의 비율이 증가할수록 범위 질의의 비율도 증가하므로 분리 저장 구조에서는 두 개 이상의 세그먼트를 검색할 확률이 높아진다. 따라서 시간지원 질의가 증가할수록 사용자 질의에 대한 성능이 저하된다.

(그림 12)는 분리 저장 구조에서 시간 단위가 분이고 시간지원 질의가 10%, 20%, 30%, 40%, 그리고 50% 각각에 대해서 질의에 대한 평균 응답 시간과, 분리하지 않은 저장 구조에서 질의에 대한 응답 시간을 나타낸 것이다. 분리 저장 구조에서는 매 분마다 데이터 이동이 일어나고 분리하지 않은 기존의 저장 구조에서는 데이터 이동이 일어나지 않는다.



(그림 12) 질의에 대한 평균 응답 시간(이동 시간단위 : 분)
(Fig. 12) Average response time at minute granularity level for queries

분리 저장 구조에서 시간지원 질의가 10%이고 사용자 질의의 평균 도착 시간이 250ms인 경우에 초 단위 이동에서 질의에 대한 응답 시간보다 분 단위에서 이동을 하는 경우에 약 50% 빠른 성능을 보인다. 이것은 매 분마다 데이터를 이동하는 부하가 매 초마다 데이터를 이동하는 부하보다 작기 때문이다.

분리 저장 구조에서는 시간지원 질의의 비율이 많아질수록 사용자 질의에 대한 평균 응답 시간이 점차 감소하게 된다. 시간지원 질의가 증가할수록 범위 질의도 늘어나기 때문에 두 개 또는 세 개 세그먼트를 검색하는 회수가 많아진다. 두 개 또는 세 개 세그먼

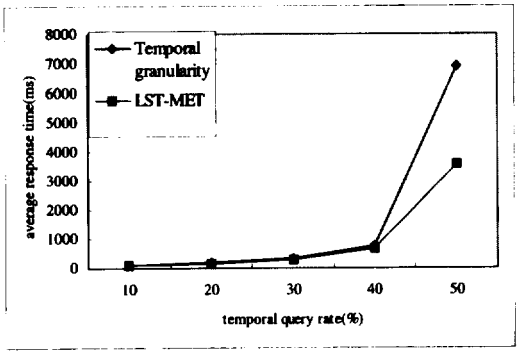
트를 검색하는 회수가 증가하면 검색 대상이 되는 튜플의 수가 늘어나므로 사용자 질의에 대한 평균 응답 시간은 감소하게 된다.

초 단위 이동과 분 단위에서 이동을 비교하면, 시간지원 질의의 비율이 많아질수록 분 단위 이동에서 성능의 감소 폭이 초 단위 이동에서 감속 폭보다 작음을 알 수 있다. 이것은 초 단위 이동에서 잦은 데이터 이동이 시스템에 주는 부하가 분 단위 이동에서 시스템에 주는 부하보다 크다는 사실을 알 수 있다.

4.3.2 데이터 이동 방법별 비교

1) LLT가 없는 경우 시간지원 질의 비율의 변화에 따른 비교

(그림 13)은 분리 저장 구조에서 시간지원 질의를 10%에서 50%까지 바꾸어 가면서 LLT의 비율이 0%인 경우에 시간 단위에 의한 이동, LST-GET에 의한 이동 방법에서 사용자 질의에 대한 평균 응답 시간을 나타낸 것이다. 이 그림의 범례에서 Temporal granularity는 시간 단위에 의한 이동 방법을 뜻하고, LST-GET는 LST-GET에 의한 데이터 이동 방법을 나타낸다. 시간단위에 의한 이동 방법에서는 시간단위를 초 단위로 한다.



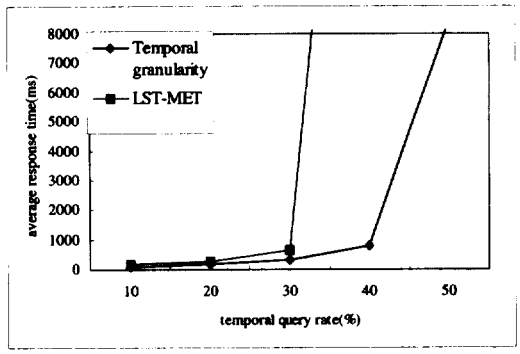
(그림 13) LLT가 없는 경우 데이터 이동 방법별 질의에 대한 평균 응답 시간

(Fig. 13) Average response time for data migration methods case in not exist LLT

(그림 13)에서 시간지원 질의가 10%인 경우에는 두 가지 데이터 이동 방법의 사용자 질의에 대한 평균 응답 시간이 거의 비슷하게 나타난다. LLT가 없으므로 두 가지 데이터 이동 방법에서 현재 세그먼트의 크기가 비슷하기 때문에 성능의 차이가 나타나지 않는다. 두 가지 이동 방법에서 시간지원 질의의 비율이 증가하면서, 세그먼트의 크기가 큰 과거 세그먼트를 접근하는 회수가 증가하므로 데이터 이동으로 인한 부하가 큰 시간단위에 의한 이동 방법의 성능이 저하된다. 이러한 현상은 시간지원 질의가 50%인 경우에 더욱 현저히 나타나서, 시간단위에 의한 이동 방법이 LST-GET에 의한 이동 방법보다 질의에 대한 성능이 약 50%로 감소하였다.

2) LLT가 있는 경우 시간지원 질의 비율의 변화에 따른 비교

(그림 14)는 LLT가 있는 경우에 분리 저장 구조에서 시간지원 질의의 비율을 바꾸어 가면서 시간단위에 의한 이동 방법과 LST-GET에 의한 이동 방법에 대해서 사용자 질의에 대한 응답 시간을 나타낸 것이다. LLT가 있으면 LST-GET에 의한 이동 방법에서는 LST가 작아지고 GET가 커지므로 현재 세그먼트에 들어갈 개체 버전을 포함하는 범위가 늘어나서 현재 세그먼트의 크기가 급격히 커진다. 반면에, 시간단위에 의한 이동 방법에서는 현재 세그먼트의 크기 변화가 비교적 작다. 현재 세그먼트를 검색하는 현재 질의가 많은 경우에, LLT가 있으면 현재 세그먼트의 크기가 커지는 LST-GET에 의한 이동 방법이, 현재 세그먼트의 크기 변화가 작은 시간단위에 의한 이동 방법보다 질의에 대한 응답 속도가 저하된다.



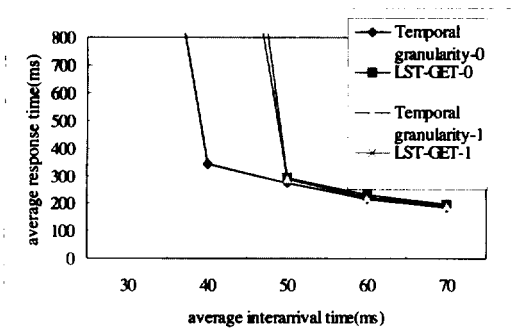
(그림 14) LLT가 있는 경우 데이터 이동 방법별 질의에 대한 평균 응답 시간

(Fig. 14) Average response time for data migration methods case in exist LLT

시간단위에 의한 이동 방법에서는 시간지원 질의의 비율이 증가하면서 범위 질의가 증가하게 되고, 따라

서 크기가 큰 과거 세그먼트를 검색하는 비율이 높아지고, 매 초마다 발생하는 데이터를 이동 부하가 크기 때문에 질의에 대한 성능의 저하를 가져왔다.

(그림 15)는 사용자 질의가 도착하는 평균 시간 간격의 변화에 대해서 시간단위에 의한 이동 방법과 LST-GET에 의한 이동 방법의 평균 응답 시간을 비교한 것이다. 그림의 범례에서 이동 방법 뒤에 붙어있는 0은 LLT가 없는 경우이고, 1은 LLT가 있는 경우를 나타낸다. LLT가 있는 경우에는 질의의 평균 도착 시간 간격이 작아짐에 따라 LST-GET에 이동 방법의 성능 저하가 가장 일찍 나타났다. 이것은 LLT가 있는 경우에는 LST-GET에 의한 이동 방법에서 현재 세그먼트의 크기가 급격히 커지기 때문에 성능의 저하가 가장 일찍 나타났다. LLT가 없는 경우에는 질의의 평균 도착 시간 간격이 작아짐에 따라 시간단위에 의한 이동 방법이 가장 오랫동안 견디다가 가장 나중에 급격한 성능의 저하를 보였다. 이것은 LLT가 없는 시간단위에 의한 이동 방법이 현재 세그먼트의 크기가 가장 작기 때문이다.



(그림 15) 질의 평균 도착 시간의 변화에 따른 평균 응답 시간

(Fig. 15) Average response time for changing average interarrival time

5. 결론 및 향후 연구 과제

이 논문에서 제안한 P-C-F 저장 구조와 데이터 이동 방법에 대해서 결론을 정리하면 다음과 같다. 이 논문에서 제안한 P-C-F 저장 구조에서는, 데이터 이동 프로세서가 데이터를 이동하기 위한 시기를 결정하는 방법과 데이터를 이동하는 과정을 제안하였다. 또한, 데이터 이동 알고리즘과 과거 세그먼트, 현재 세그먼트, 그리고 미래 세그먼트에 저장되는 개체 버전에 대해서 정의하였다. P-C-F 저장 구조를 기반으로 하는 두 가지 데이터 이동 방법을 제안하였다. 제안한 데이터 이동 방법은 시간단위에 의한 이동 방법과 LST-GET에 의한 이동 방법이 있다.

시간단위에 의한 이동 방법에서 각 세그먼트에 저장되는 개체 버전을 정의하고 각 세그먼트에서 데이터의 유효성을 검사하고 이동하는 과정을 보였다. 시간단위에 의한 이동 방법에서 데이터 이동에 따른 부하가 지나치게 커지는 경향을 개선하기 위해서 LST-GET에 의한 이동 방법을 제안하였다. LST-GET에 의한 데이터 이동 방법에서 LST와 GET를 정의하고 경계값과 유효 시간과의 관계를 통해서 각 세그먼트에 저장되는 개체 버전을 정의하였다. LST와 GET를 구하는 방법을 제시하고 LST와 GET를 기준으로 이동 대상이 되는 개체 버전을 검색하고 이동하는 과정을 보였다.

기존의 분리하지 않은 저장 구조와 이 논문에서 제안한 P-C-F 저장 구조에 대한 성능 비교와, 이 논문에서 제안한 네 가지 데이터 이동에 대한 성능을 분석한 결과를 요약하면 다음과 같다. P-C-F 저장 구조에서 시간단위에 의한 이동 방법을 쓰고 초 단위와 분 단위로 데이터 이동이 발생하는 경우에 대해서, 기존의 분리하지 않은 저장 구조에서 질의를 처리하는 것보다 우수한 성능을 보였다. LLT가 없는 경우에는 시간단위에 의한 이동 방법과 LST-GET에 의한 이동 방법 중에서 데이터 이동 부하가 큰 시간단위에 의한 이동 방법의 성능이 저하되었다. LLT가 있는 경우에는 현재 세그먼트의 크기가 커지는 LST-GET에 의한 이동 방법의 성능이 저하되었다.

본 연구와 관련해서 앞으로 연구할 과제는 다음과 같다. 이 논문에서 제안한 P-C-F 저장 구조를 설계하고 실제로 구현하는 것이 필요하다. 미래 세그먼트에 들어있는 개체 버전은 변경될 가능성이 있으므로 변경된 개체 버전을 관리하는 다른 세그먼트를 고려한 연구가 추가로 필요하다. 이 논문에서 제안한 데이터 이동 방법을 P-C-F 저장 구조상에서 실제 구현해 봄으로써 검증해 보는 것이 필요하다.

참 고 문 헌

[1] I. Ahn and R.T. Snodgrass, "Partitioned Storage for Temporal Databases," Information Systems,

Vol.13, No.4, pp.369-391, 1988.

[2] M.H. Bohlen, "Temporal Database System Implementation," ACM SIGMOD Record, Vol.24, No.4, pp.53-60, December 1995.

[3] C.E. Dyreson and R.T. Snodgrass, "Valid-time Indeterminacy," Proceedings of the International Conference on Data Engineering, pp.335-343, Vienna, Austria, April 1993.

[4] R. Elmasri, G.T.J. Wuu and Y. Kim, "The Time Index : An Access Structure for Temporal Data," Proceedings of the 16th Conference on Very Large Databases, Brisbane, Australia, August 1990.

[5] O. Etzion, A. Gal, and A. Segev, "Retroactive and Proactive Database Processing," ftp://segev.lbl.gov/pub/94-Rideb.ps

[6] S.K. Gadia, and C.S. Yeung, "A Generalized Model for a Relational Temporal Database," Proceedings of the ACM International Conference on Management of Data, pp.251-259, Chicago, IL, June 1988.

[7] C.S. Jensen, R. Elmasri, S.K. Gadia, P. Hayes, and S. Jajodia(eds), "A Glossary of Temporal Database Concepts," ACM SIGMOD Record, Vol. 23, No.1, pp.52-64, March 1994.

[8] C.S. Jensen and R.T. Snodgrass, "Temporal Specialization and Generalization," IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.6, pp.954-974, December 1994.

[9] C.S. Jensen, M.D. Soo, and R.T. Snodgrass, "Unifying Temporal Models via a Conceptual Model," Information Systems, Vol.19, No.7, pp. 513-547, December 1994.

[10] S. Kim and S. Charkravarthy, "Temporal Databases with Two-Dimensional Time : Modeling and Implementation of Multihistory," Information Sciences, Vol.80, nos.1-2, pp.43-89, September 1994.

[11] C. Kolovson, and M. Stonebraker, "Segment Indexes : Dynamic Indexing Techniques for Multi-Dimensional Interval Data," Proceedings of ACM SIGMOD Conference on Management of Data, pp.138-147, June 1991.

[12] V. Kouramajian, "Temporal Databases : Access Structures, Search Methods, Migration Strategies, and Declustering Techniques," Ph. D. Dissertation, The University of Texas at Arlington, 1994.

[13] M.A. Nascimento, M.H. Dunham, and R. Elmasri. "M-IVTT : A Practical Index for Bitemporal Databases," Proceedings. of the 7th International Conference on Database and Expert Systems Applications(DEXA'96), September 1996.

[14] S.B. Navathe and R. Ahmed, "A Temporal Relational Model and a Query Language," Information Sciences, Vol.49, pp.147-175, 1989.

[15] B. Salzberg and V.J. Tsotras, "A Comparison of Access Methods for Temporal Data," Technical Report(TR-18), TimeCenter, June 1997.

[16] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R.T. Snodgrass(eds.), Temporal Databases : Theory, Design, and Implementation, Database Systems and Applications Series, Chapter 5, Benjamin/cummings, Redwood City, CA, 1994.

[17] R.T. Snodgrass, "The Temporal Query Language Tquel," ACM Transactions on Database Systems, Vol.12, No.2, pp.247-298, June 1987.

[18] R.T. Snodgrass, M.H. Bohlen, C.S Jensen, and A. Steiner. Adding Valid Time to SQL/Temporal. ANSI Expert's Contribution, ANSI X3H2-96-501r1, ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2, International Organization for Standardization, November 1996, pp.77.

[19] R.T. Snodgrass, M. Bohlen, C.S. Jensen, and A. Steiner, Adding Transaction Time to SQL/Temporal, ANSI Expert's Contribution, ANSI X3H2-96-502r2, ISO/IEC JTC1/SC21/WG3 DBL MCI-147r2, International Organization for Standardization, October 1996, pp.47.

[20] R.T. Snodgrass, M. Bohlen, C.S. Jensen, and A. Steiner, "Transitioning Temporal Support in TSQL2 to SQL3," Technical Report(TR-8), Time-Center, March 1997.

[21] A.U. Tansel, "A Historical Query Language," Information Sciences, Vol.53, pp.101-133, 1991.

[22] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R.T. Snodgrass(eds.), Temporal Databases : Theory, Design, and Implementation, Database Systems and Applications Series, Benjamin/Cummings, Redwood City, CA, 1994.

[23] J. Won and R. Elmasri, "Representing Retroactive and Proactive Versions in Bi-Temporal Data-Bases," Proceedings on 12th International Conference on Data Engineering, pp.85-94, 1996.

[24] H. Yun and K. Kim, "Experimenting with Segmentation and Non-segmentation Methods for Storing Temporal Data," Proceeding of CNDS'98, San Diego, CA, pp.113-118, January 1998.

[25] 박유현, 이중화, 윤홍원, 김경석, "분리 저장 구조를 가지는 시간지원 데이터베이스에서 데이터 이동 기법에 관한 연구", 한국정보과학회, '97가을학술발표논문집, 24권 2호, pp.469-472, 1997. 10.

[26] 윤홍원, 김경석, "시간지원 데이터베이스에서 자료의 이동을 고려한 저장 방법의 성능 평가", 한국정보과학회논문지, 25권 5호, pp.756-767, 1998. 5.

[27] 윤홍원, 정연정, 김경석, "시간 자료의 특성을 고려한 분리 저장 방법", 한국정보과학회, '97봄학술발표논문집, 24권1호, pp.109-112, 1997. 4.

윤 홍 원

e-mail : hwyun@lotus.silla.ac.kr

1986년 부산대학교 계산통계학과 졸업(학사)

1990년 한국외국어대학교 경영정보대학원 전자계산학과(이학석사)

1998년 부산대학교 대학원 전자계산학과(이학박사)

1996년~현재 신라대학교 컴퓨터정보공학부 조교수

관심분야 : 데이터베이스 시스템, 시간지원 데이터베이스, 멀티미디어 데이터베이스, 의료 정보 시스템, 데이터 웨어하우징

김 경 석

e-mail : gimgs@asadal.cs.pusan.ac.kr

1977년 서울대학교 무역학과 졸업(학사)

1979년 서울대학교 대학원 계산통계학과(이학석사)

1988년 미국 일리노이(어바나 샴페인) 주립대학교(전산학박사)

1988년~1992년 미국 노스타코타 주립대학교 전산학과 조교수

1992년~현재 부산대학교 전자계산학과 부교수

관심분야 : 데이터베이스 시스템, 멀티미디어 시스템, 한글 정보 처리, 음성 인식