Hashed B-트리 인덱스를 이용한 효율적인 무결성 검사 (Efficient Integrity Checking Using Hashed B-Tree Index)

요약문

본 논문에서는 무결성 제약을 효율적으로 유지하기 위한 접근 경로인 Hashed B-트리를 제안하고, 기존의 B-트리와 성능을 비교한다. 무결성 제약을 만족시키기 위해 필요한 질의 패턴들이 종점 질의로 구성되어 있다는 점에 착안하여, Hashed B-트리는 키를 해싱을 통해 압축하여 저장한다. 따라서 트리의 높이가 줄어들게 되고 트리의 탐색이 빠르다. 이 기법은 기존의 B-트리와 유사하게 작동하기 때문에 기존 시스템에 많은 수정을 요하지 않고 병행성 제어나 회복 기법을 그대로 사용할 수 있는 장점이 있다.

주제어 : 데이타 베이스, 무결성 제약, 접근 경로, B-트리

Abstract

This paper suggests a new access path, hashed B-tree which is an efficient access method for integrity checking. Hashed B-tree is based on the observation that most query patterns in enforcing integrity constraints are point queries. Hashed B-tree compresses the key by hashing procedure, which reduces the height of tree and results in fast node search. This method has the advantages such as it can be implemented easily and use the B-tree concurrency control and recovery algorithm with minor modifications.

Keyword: database, integrity constraint, access path, B-tree

1 서론

1.1 연구 배경 및 필요성

무결성 제약 조건은 데이타베이스를 의미적으로 오류가 없는 상태로 유지하기 위해 만족시켜야 하는 규칙이다. 무결성을 각각의 응용 프로그램에서 보장하기 위해서는 입력되는 데이타가 올바른 값의 범위에 드는지, 또는 두 개 이상의 값들간의 관계가 올바른지 등을 지속적으로 검사해 주어야 한다. 여러 응용 프로그램이 데이타를 공유함에 따라, 무결성을 보장하기 위한 응용 프로그램의 코드는 점점 복잡해지고 유지 관리하기가 힘들어진다. 따라서 여러 응용 프로그램에 공통으로 적용되는 무결성 제약 조건을 데이타 정의어(data definition language)나 데이타 제어어(data control language)로 명시하면 데이타베이스 시스템이 이를 자동으로 보장해 주는 방식을 취하게 되었다.

표준 SQL에서는 테이블 전체에 적용되는 제약 조건, 테이블의 특정 열(column)에 적용되는 제약 조건, 기본 키(primary key) 제약 조건, 외래 키(foreign key) 제약 조건, 참조 무결성(referential integrity) 등을 지원할 것을 명시하고 있다 [8, 15].

다양하고 복잡한 응용들이 등장하면서 데이타베이스 시스템이 보장해줘야 하는 이러한 무결성에 대한 요구가 많아지게 되었고, 이를 효율적으로 지원해야할 필요성이 커지게 되었다. 이와 같이 무결성 제약은 데이타베이스의 설계나 응용에 있어 필수적인 요소로 인식되고 있으나 데이타베이스가 이러한 제약들을 실제로 만족하고 있는가의 평가는 복잡할 뿐만아니라 처리 시간 등에 있어 많은 비용이 요구되는 어려운 일이다 [1]. 예를 들어 참조 무결성을 유지하기 위해서는, 무결성 제약 조건에 직접적으로 영향을 받는 테이블들에 대하여 삽입, 삭제, 갱신 등의 연산을 수행할 때마다 참조 관계에 있는 튜플들을 찾아서, 이들간의 참조 관계가 올바른지 검사하여 적절한 동작을 취해주어야 한다. 또한 이러한 검사를 위해서는 다량의 데이타를 접근해야 되므로 매우 큰 비용이 들게 된다. 그러므로, 빈번히 접근되는 데이타에 대한 부하를 줄이기 위해서 효율적인 접근 방법(access method)이 필요하다.

Härder [14]는 부모 릴레이션과 자식 릴레이션 사이의 참조 무결성을 유지하기 위한 기능적 요구 조건(functional requirements)을 정리하면서, 모두 종점 질의[1]로 구성되어 있는 무결성 유지를 위한 연산들을 가장 효과적으로 만족시킬 수 있는 접근 경로가 B-트리[2]임을 보였다. 그런데, B-트리에 해싱 기법을 적용시키면 종점 질의에 대해 트리를 탐색해야 하는 깊이를 줄여, 보다 유용하고 효율적인 접근 경로를 만들 수 있다.

본 논문에서는 무결성 제약을 효율적으로 지원하기 위한 새로운 접근 경로인 Hashed B-트리(이하 HB-트리)를 제안하고 그 성능을 평가한다.

1.2 논문의 구성

2절에서는 무결성 제약, 다양한 키 압축 기법, 인덱싱 기법에 대한 관련 연구를 살펴 본다. 3절에서는 무결성 제약을 지원하기 위한 접근 경로의 특징과 비용에 대해 설명하고, 4절에서는 본 논문에서 무결성 제약을 효율적으로 지원하기 위해 제안한 접근 경로인 HB-트리의 구조와 특징, 구현에 대해 설명한다. 5절에서는 HB-트리와 기존의 B-트리의 성능을 분석 비교하고, 6절에서 결론에 대해서 이야기한다.

2 관련 연구

이 절에서는 본 논문의 관련 연구에 대해 살펴 본다. 먼저 무결성 제약 조건을 유지하기 위해 드는 비용을 분석한 관련 연구를 살펴 보고, 다음으로 B-트리의 성능을 향상시키기 위한 연구들을 살펴 본 후, 해싱 기법에 대해 알아 본다.

[1] 종점 질의에 상대되는 개념은 범위 질의(range query)이다.

[2] 대부분의 시스템에서 구현되는 B^+-트리 또는 B^*-트리를 단순히 B-트리라고 통칭함.

2.1 무결성 제약 조건

Codd [9]는 관계형 데이타 모델에서 개체 무결성(entity integrity)와 참조 무결성을 소개하고 무결성은 시스템에서 자동적으로 보장해주어야 한다고 말한다. 관계형 데이타베이스 시스템에서 지원해야 하는 이러한 무결성 제약의 의미를 일반적으로 정의하고 정형화하기 위한 노력이 지속적으로 진행되어 왔다 [10, 12]. 이를 바탕으로 무결성을 유지하는 비용과 성능을 분석하고자 하는 연구가 시도되었다.

Badal [3]은 무결성 제약을 단언하는 시점을 트랜잭션의 수행 순서에 연관하여 컴파일시, 실행시, 실행 후로 분류하고 각각의 비용을 분석한다. 여기서는 비용을 평가하는 주된 요소로 계산 시간과 보조 기억 장치에 저장되어 있는 데이타베이스 데이타를 접근하는 회수를 들고 있다.

Härder [14]는 연산 단위로 데이타베이스 시스템의 행태를 관찰함으로써 보다 세부적으로 무결성 개념이 데이타베이스 시스템 성능에 미치는 영향을 알아보고자 하였다. 시스템의 부하에 영향을 미치는 것은 질의 수행시에 빈번히 행해지는 원하는 튜플(tuple)들을 찾고, 그것의 키값을 검사하고 비교하는 다양한 탐색들을 들 수 있다. 또한 탐색한 튜플들을 갱신해야 할 때[3], 튜플의 갱신 뿐아니라 그에 따른 접근 경로의 갱신도 이루어져야 한다. 그 이외에도 로킹(locking), 로깅(logging)에 따른 부가적인 부하가 발생하는데, 주로 기존의 다양한 접근 경로를 탐색할 때 생기는 부하들을 페이지 참조 회수에 의거하여 평가하고 있다.

2.2 B-트리 키 압축 기법

B-트리는 상용 데이타베이스 시스템에서 널리 사용되고 있는 인덱싱 기법 중의 하나이다. 따라서 B-트리의 구성과 이를 통한 탐색은 데이타베이스 관리 시스템(DBMS)의 성능을 결정짓는 중요한 요소이므로 B-트리의 성능을 향상시키기 위한 다각도의 연구가 진행되어 왔다.

이러한 연구의 목적은 주로 B-트리 저장 공간의 절약과 탐색시의 디스크 접근 회수를 줄이는 것이라 말할 수 있다. 이를 위해 가변 길이의 키를 지원하고 효율적인 키 압축 기법을 지원하고자 하는 노력이 계속되어져 왔다 [2, 4, 7]. B-트리에 저장되는 키를 압축하게 되면 키를 정렬하고 탐색할 때 걸리는 시간을 절약할 수 있으며, 또한 디스크의 접근 회수를 감소시키고 저장 공간의 효율성을 꾀할 수 있다.

Bayer [4]는 B-트리의 비단말 노드에 키 전체를 저장하는 것이 아니라 다른 키들과 구분될 수 있는 구분자의 역할이 가능한 키의 앞부분(prefix)만을 저장하는 후미 압축 기법을 제안하였다. 이 기법은 비단말 노드에 저장되는 키의 크기를 줄임으로써 B-트리 탐색을 빠르게 해주며 B-트리 노드들의 전개율(fanout)을 높여주나, 인덱스를 유지하기 위해 비단말 노드의 구분자를 동적으로 계속 재구성해야 하는 단점이 있다.

System R에서는 다중 필드 키를 위한 압축 기법을 고안하여 구현하였다 [7]. 다중 필드 키를 압축할 때, 각각의 필드들을 압축한 후 이들을 접합(concatenation)시키면 정상적으로 비교가 되지 않는 단점이 생기고, 모든 필드들을 고정 크기로 한다면 부가의 문자를 삽입해야 되므로 디스

[3] 참조 무결성 유지를 위해 발생한다.

크의 낭비가 생기게 된다. System R에서는 제어 문자를 압축시킬 필드 사이에 삽입시킴으로써 이러한 문제들을 해결하였다. 그러나 여기서 사용된 키 비교 방법이 SQL에서의 명시된 공백 문자(blank)의 처리 방법과 다르다는 단점이 있다 [2].

순서 유지(order preserving) 디렉토리 압축 기법 [2]은 보다 강력한 데이타 압축 기법을 사용하면서도 키의 순서를 유지하는 기법으로 압축된 키와 압축되기 이전의 키들의 관계를 사전(dictionary or table)에 유지하여, 다중 필드 키를 단축함에 있어 System R에서 발생한 문제점을 개선하였다. 하지만 부가적인 자료구조가 필요하고 알고리즘이 복잡하다는 단점이 있다.

위의 관련 연구들의 공통적인 특징은 B-트리 탐색시 범위 질의를 가능하게 하기 위하여 키의 순서를 유지하고 있다는 것이다. 그러므로, 3절에서 보이겠지만, 인덱스의 용도가 참조 무결성 조건의 검사에서처럼 범위 질의를 필요로 하지 않는다면 보다 효과적인 압축 기법을 고려할 수 있게 될 것이다.

2.3 해싱

트리와 해싱 기법을 접목시키려는 시도를 찾아볼 수 있다 [5, 6, 11]. 해싱 방법은 다른 레코드를 참조하지 않고 목표 레코드를 직접 접근할 수 있게 하는 기법이다. 보통 레코드를 식별하기 위한 키 값과 보조 기억 장치에 저장되어 있는 레코드 주소 사이의 관계를 디렉토리(directory)라는 자료 구조로 유지한다. 이 디렉토리를 트리의 형태로 만들어 데이타를 직접 접근할 뿐아니라 순차적으로 접근할 수 있도록 하고, 키 공간의 축약을 통해 저장 공간의 유용성을 높이려는 연구 [5, 6, 11]가 있었다.

3 무결성 제약 지원을 위한 접근 경로

2.1절에서 언급했듯이 Härder [14]는 기존의 다양한 접근 경로들을 참조 무결성을 지원하기 위해 사용할 때 얼마나 유용한가에 대해 밝히고 있다. 이 절에서는 참조 무결성의 의미[4]와 접근 경로에 대한 접근 패턴에 대해 설명한다.

3.1 참조 무결성

참조 무결성은, 개체 무결성과 더불어 관계형 데이타 모델의 핵심 요소로서, Codd [9] 및 Date [10] 등에 의해 개념이 정립되어, 현재는 관계형 데이타베이스 표준에 포함되어 있다.

참조 무결성이란, 특정 릴레이션 R_P의 기본 키 PK와 이 기본 키를 참조하는 다른 릴레이션 R_C[5]의 외래 키 사이에 반드시 만족되어야 하는 조건으로, R_C의 임의의 튜플 t의 $t[FK]$값은 널($null$)값이거나 R_P에 같은 값을 기본 키로 갖는 튜플이 반드시 존재해야 한다.

[4]무결성 제약 조건에 대한 비용 계산을 참조 무결성에 국한시키는 이유는 참조 무결성을 유지하기 위해서는 기본 키 제약 조건이나 외래 키 제약 조건, 유일성(unique) 제약 조건 등이 병행적으로 수행되기 때문이다.

[5]R_P와 R_C는 각각 부모 릴레이션(parent relation), 자식 릴레이션(child relation)을 나타낸다.

참조무결성 1 (SQL2에서의 참조 무결성) $t \in R_C$, $t[FK] \neq null$인 튜플 t에 대해, R_P에는 $t[FK] = q[PK]$인 하나의 튜플 q가 반드시 존재한다.

다음은 SQL2에서 참조 무결성의 선언과 관련한 구문을 보여주고 있는데, 데이타 정의어를 이용해서 R_C의 선언시에 추가적으로 외래 키와 참조 무결성의 동적 의미를 정의하는데 사용된다.

```
FOREIGN KEY (<referencing columns>)
        REFERENCES <table name> (<referenced columns>)
      [ON UPDATE { CASCADE | SET NULL | SET DEFAULT | NO ACTION } ]
      [ON DELETE { CASCADE | SET NULL | SET DEFAULT | NO ACTION} ]
```

위의 구문에서 `<referencing columns>`은 R_C의 외래 키 FK를 나타내고, `<table name>`은 참조되는 R_P를 나타낸다. SQL92에서는 `<referenced columns>`에 R_P의 후보 키(candidate key)도 지정할 수 있으나, 본 논문에서는 항상 기본 키로 가정한다. 마지막 옵션은 R_P에서 특정 튜플 t의 기본 키 값 $t[PK]$가 변경되거나 t가 삭제될 때, R_C에서 이를 참조하고 있는 튜플들을 어떻게 처리할 것인가를 명시하는데 사용된다. 예를 들어, R_P의 한 튜플 t가 삭제될 때 `CASCADE`는 참조 무결성 조건을 만족시키기 위해 R_C에서 $t'[FK] = t[PK]$인 모든 튜플 t'를 삭제한다. `SET NULL`의 경우에는 $t[PK]$값을 참조했던 모든 튜플 t'들의 외래 키 값을 모두 널로 바꾸게 된다. `SET DEFAULT`의 경우에는 $t[PK]$값을 참조했던 모든 튜플들의 외래 키 값을 모두 R_C의 디폴트값으로 바꾸게 된다.

3.2 접근 경로상의 비용

참조 무결성 관계를 갖는 부모 릴레이션과 자식 릴레이션을 각각 P, C라고 할 때, 접근 경로에 대한 질의[6] 형식은 다음과 같이 정의할 수 있다.

$(P, E)_{\mathrm{P}}$: 부모 릴레이션(P)에 키의 존재(Existence) 여부를 검사하기 위한 종점 질의(Point query)

$(P, T)_{\mathrm{P}}$: 부모 릴레이션(P)에서 조건을 만족하는 튜플을 찾는 종점 질의

$(P, E)_{\mathrm{C}}$: 자식 릴레이션(C)에 키의 존재 여부를 검사하기 위한 종점 질의

$(P, T)_{\mathrm{C}}$: 자식 릴레이션(C)에서 조건을 만족하는 튜플(Tuple)을 찾는 종점 질의

$(P, S)_{\mathrm{C}}$: 자식 릴레이션(C)에서 조건을 만족하는 튜플들의 집합(Set)을 찾는 종점 질의

참조 무결성 제약 조건을 유지하기 위해 필요한 질의 비용을 릴레이션에 적용되는 연산에 따라 간략히 설명하면 다음과 같다. 보다 자세한 비용 계산 방법은 [14]을 참조하면 된다.

[6]여기서의 질의는 질의어 처리 과정 단계의 질의를 말하는 것이 아니라, 접근 경로를 참조하고 유지하기 위해 사용되는 질의를 말한다.

3.2.1 부모 릴레이션에의 삽입 연산

참조 무결성을 위배하지 않는 연산이다. 삽입 전에 부모 릴레이션의 기본 키 제약 조건의 만족 여부를 조사해야 한다. 부모 릴레이션에 삽입될 키가 이미 기본 키로 존재하는지 여부를 검사하는 질의 $(P, E)_{\mathrm{P}}$가 필요하다.

3.2.2 부모 릴레이션에의 삭제 연산

부모 릴레이션에서 튜플을 삭제할 경우 먼저 해당 자식 튜플이 자식 릴레이션에 존재하는지 검사한다. 접근 경로에서 튜플의 위치를 찾는 이 질의의 형식은 $(P, T)_{\mathrm{P}}$이다. 이 때, 삭제될 튜플을 참조하고 있던 자식 릴레이션의 튜플이 존재할 수 있다. 삭제 연산 직후 이 자식 릴레이션의 튜플들이 참조할 더 이상의 부모 튜플이 존재하지 않으면 참조 무결성을 위배하게 된다. 참조 무결성의 옵션에 따라 이들을 검사하고 유지하는데 필요한 질의 형식은 다음과 같다.

CASCADE : 삭제될 부모 튜플을 참조하고 있는 자식 튜플들도 삭제시킨다. 따라서, 총 $(P, T)_{\mathrm{P}} + (P, S)_{\mathrm{C}}$의 질의가 필요하다.

RESTRICT : 삭제될 부모 튜플을 참조하는 자식 튜플이 하나라도 존재하면 삭제 연산을 수행하지 못하도록 해야 한다. 따라서 $(P, T)_{\mathrm{P}} + (P, E)_{\mathrm{C}}$의 질의가 필요하다.

SET NULL : 삭제될 부모 튜플을 참조하는 참조하는 자식 튜플들을 널값으로 바꾼다. 자식들을 찾고, 이를 접근 경로 상에서 갱신해야 하는 질의가 필요하다. 따라서 비용은 $(P, T)_{\mathrm{P}} + 2{\cdot}(P, S)_{\mathrm{C}}$로 정리할 수 있다.

SET DEFAULT : 삭제될 부모 튜플을 참조하는 자식 튜플들을 디폴트값으로 바꾼다. SET NULL과 다른 점은 갱신되어야 할 디폴트값은 널값과 달리 기본 키 제약을 만족시켜야 한다는 것이다. 따라서 디폴트값이 부모 릴레이션에 있는지 검사하는 질의가 추가되어 $(P, T)_{\mathrm{P}} + (P, E)_{\mathrm{P}} + 2{\cdot}(P, S)_{\mathrm{C}}$의 비용이 든다.

3.2.3 부모 릴레이션에의 갱신 연산

갱신 연산은 이전 튜플에 대한 삭제 연산과 새로운 값을 갖는 튜플의 삽입 연산의 조합으로 처리될 수 있다. 최악의 경우 $(P, T)_{\mathrm{P}} + 2{\cdot}(P, E)_{\mathrm{P}} + 2{\cdot}(P, S)_{\mathrm{C}}$의 비용이 든다.

3.2.4 자식 릴레이션에의 삽입 연산

부모 릴레이션에 참조 무결성을 만족하는 부모 튜플이 존재하는지 검사해보는 $(P, E)_{\mathrm{P}}$ 질의와 접근 경로에서 삽입될 위치를 찾는 $(P, T)_{\mathrm{C}}$ 질의가 필요하다.

3.2.5 자식 릴레이션에의 삭제 연산

참조 무결성을 위배하지 않는 연산이다. 자식 릴레이션에 삭제 연산을 수행할 때 접근 경로를 통해 삭제될 튜플을 검색한다. $(P,T)_{\mathrm{C}}$의 연산이 필요하다.

3.2.6 자식 릴레이션에의 갱신 연산

갱신할 자식 튜플을 찾고, 부모 릴레이션에 참조 무결성을 만족하는 부모 튜플이 존재 여부를 검사해보고, 접근 경로를 갱신한다. 총 $(P,E)_{\mathrm{P}} + 2{\cdot}(P,T)_{\mathrm{C}}$의 질의가 필요하다.

참조 무결성을 유지하기 위해 접근 경로를 탐색할 때 필요한 질의 비용을 정리하면 다음과 같다.

연산	비용
부모 릴레이션에 삽입	$(P,E)_{\mathrm{P}}$
부모 릴레이션에 삭제	$(P,T)_{\mathrm{P}} + (P,E)_{\mathrm{P}} + 2{\cdot}(P,S)_{\mathrm{C}}$
부모 릴레이션에 갱신	$(P,T)_{\mathrm{P}} + 2{\cdot}(P,E)_{\mathrm{P}} + 2{\cdot}(P,S)_{\mathrm{C}}$
자식 릴레이션에 삽입	$(P,E)_{\mathrm{P}} + (P,T)_{\mathrm{C}}$
자식 릴레이션에 삭제	$(P,T)_{\mathrm{C}}$
자식 릴레이션에 갱신	$(P,E)_{\mathrm{P}} + 2{\cdot}(P,T)_{\mathrm{C}}$

지금까지 살펴본 바와 같이 참조 무결성 검사를 위해 필요한 질의 형태는 모두 종점 질의이다. 또한 개체 무결성 검사에도 종점 질의만 쓰인다. 많은 경우 B-트리를 사용하여 기본 키와 외래 키에 대한 무결성 검사를 수행하는데, 종점 질의를 처리할 때에는 B-트리의 순서 기능은 사용되지 않는다. 이처럼 순서를 유지할 필요가 없는 상황에서는 기존의 압축 기법보다 훨씬 효과적인 압축 기법을 고려해 볼 수 있는데 본 논문에서는 해싱을 압축 기법으로 사용하고자 하는 것이다.

4 HB-트리

4.1 구조

무결성 제약을 지원하기 위한 접근 경로는 3절에서 보았듯이 오직 종점 질의의 용도로만 사용되기 때문에 범위 질의를 위해 요구되는 순서 유지가 필요치 않다. 따라서 HB-트리는 B-트리를 구성할 때, 키값을 해싱(hashing)을 통하여 해쉬키(hash key)로 변환하고 트리의 비단말 노드에 이 해쉬키를 저장한다. 그리고 실제 키값은 단말 노드에만 저장시킨다.

그림 1에서 보면, 원하는 튜플을 인덱스를 통해 찾을 때에는 탐색키를 인덱스를 구성할 때 사용했던 동일한 해쉬 함수를 적용시켜 해쉬키로 변환하고, 이 해쉬키로 B-트리를 탐색하여 단말 노드를 찾아서 동일한 해쉬키를 갖는 키값들을 비교하여 원하는 튜플의 레코드 식별자를 얻게 된다. 그러면 레코드 식별자를 통해 데이타베이스 내에 저장되어 있는 튜플을 접근하게 되는 것이다.

HB-트리의 삽입, 삭제 알고리즘은 기존의 B-트리 삽입, 삭제 알고리즘과 동일하다. 단지 탐색 시 해쉬키를 비교하는 점이 차이가 날 뿐이다. 5.1절에 본 논문에서 구현된 HB-트리의 삽입, 삭제 알고리즘을 기술한다.

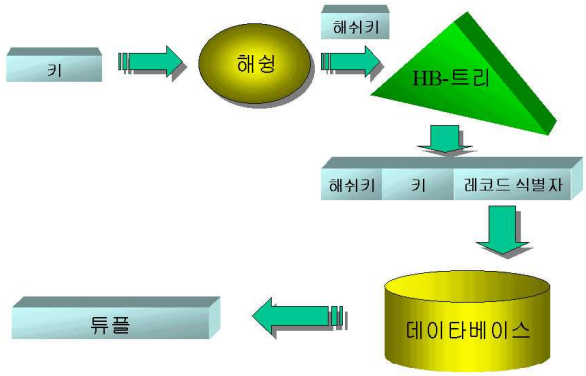

그림 1: HB-트리의 구조

HB-트리의 단말 노드에 저장되는 레코드의 양식은 그림 2와 같이 여러 가지 방법으로 구성할 수 있다.

양식 1 : 인덱스 내에 전체 튜플을 저장하는 구조이다. 이 구조는 동일한 해쉬키를 갖는 튜플들이 물리적으로 집중되어 있으므로 실제 튜플을 얻기 위하여 레코드 식별자들을 통해 디스크에 분산되어 있는 튜플들을 접근해야 하는 과정을 제거하는 장점이 있으나, 인덱스가 매우 커지게 되어 저장 공간의 소모가 크며 저장 시스템에 저장된 튜플과 인덱스에 저장된 튜플 간의 일관성을 보장하기 위해 잦은 갱신이 필요하다는 단점이 있다.

양식 2 : 해쉬키와 그 해쉬키를 갖는 튜플의 레코드 식별자들을 저장한다. 이 구조는 단말 노드에 저장되는 데이타의 크기가 가장 적은 장점이 있다. 그러나, 해쉬키를 생성할 때 충돌이 발생한다면 레코드 식별자를 통해 튜플을 접근한 후 다시 실제 키값들의 비교를 통해 탐색하고자 하는 레코드인가를 확인해야 하는 부하가 발생한다. 그러므로 해쉬 함수가 효율적이지 못하면 필요없는 페이지 접근이 많게 된다. 만약 해쉬 함수에 의한 충돌(collision)의 발생율이 낮다면 실제키를 다시 비교해야 하는 부하를 제거할 수 있어 가장 효율적인 구조이다.

양식 3 : 해쉬키순으로 정렬하고, 동일 해쉬키를 갖는 실제 키값과 레코드 식별자들의 쌍을 저장하는 구조이다. 양식 1과 양식 2를 절충한 형태로써 실제 튜플을 접근하지 않고 실제 키값의 비교가 가능하다.

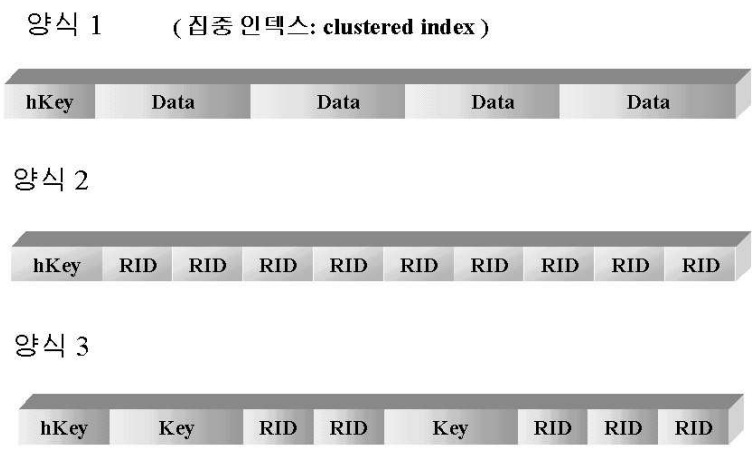

그림 2: 단말 노드의 구조

4.2 특징

HB-트리의 특징들을 살펴보면 다음과 같다.

HB-트리는 기존의 B-트리가 구현되어 있는 시스템에 큰 수정없이 구현될 수 있다. 따라서 코드의 재사용성이 높다. 즉, 트리를 구성하고 유지하는 기법은 전혀 변화가 없으며, 트리에 저장될 키와 단말 노드의 구조가 바뀔 뿐이다. 또한 기존의 B-트리에서 사용되고 있는 병행성 제어 기법이나 회복 기법 [17–19]을 큰 수정없이 사용할 수 있다는 장점이 있다.

HB-트리에서는 탐색키를 해쉬 함수를 통해 해싱을 시킴으로써 키의 크기를 줄일 수 있다. N 바이트의 키를 해쉬 함수를 이용해 K 바이트의 해쉬키로 만들었을 때 트리의 비단말 노드에서는 N/K 배의 저장공간 효율성을 얻게 되는 것이다. 이는 탐색키가 가변 길이를 갖는 다중 필드의 키일 경우에도 동일한 압축의 효과를 볼 수 있다. 즉, 원래 키의 길이와는 무관하게 고정 길이의 키값으로 전환 가능하다. 이처럼 트리 인덱스의 비단말 노드를 구성하는 키를 압축시킴으로써 트리의 높이가 줄어들게 된다. 따라서 트리를 탐색하기 위해 필요한 페이지 접근 회수가 줄어들고, 트리의 전개율이 높아진다. 이는 인덱스 성능 향상에 크게 기여함을 5절에서 보인다.

여러 트랜잭션이 동일한 테이블에 레코드를 순서에 따라 입력할 경우 이를 병렬적으로 수행할 수 있다. 만약 이 경우 테이블에 인덱스가 만들어져 있으면 인덱스도 함께 구성을 해주어야 한다. B-트리는 키들을 순차 입력할 경우 트리의 루트 노드에서 단말 노드에 이르는 경로에 대한 잠금(lock)을 계속적으로 갖게 되어 다른 트랜잭션이 접근하지 못하는 병목 현상이 발생하게 된다. 그러므로 하나의 트랜잭션이 종료하기까지 다른 트랜잭션은 수행할 수 없어 병렬적으로 테이블에 레코드를 입력할 수 없게 되는 것이다. HB-트리는 해싱을 통해 순차적인 키들을 분산시켜 입력함으로써 B-트리에서 발생하는 병목 현상을 해결할 수 있다.

HB-트리에서 해쉬키를 만들 때, 해쉬 함수의 특성에서 기인하는 충돌이 발생한다. 따라서 서로 다른 탐색키가 충돌에 의해 동일한 해쉬키를 갖게 될 때 이를 해결하기 위한 부가의 비교 연산이 단말 노드에서 수행되어져야 한다. 하지만 해쉬 함수가 완전(perfect)하다면 이러한 단점은 없

어지게 된다. 해쉬키 값의 범위를 크게 하면 충돌 가능성을 줄일 수 있는데 예를 들어 64 비트 해쉬키를 만들면 2^{64}의 키 공간을 갖게 된다.

HB-트리가 갖고 있는 또다른 단점은 탐색키를 해쉬키로 변환하여 트리에 저장함으로써 범위 질의를 수행하지 못한다는 것이다. 하지만 무결성 제약을 지원하기 위한 접근 경로는 범위 질의를 요구하지 않음을 3절에서 설명하였다. 또한 해쉬 함수를 순서가 유지되도록 잘 만들 수 있다면[7] HB-트리에 범위 질의를 지원할 수도 있으므로 이 단점은 보완할 수 있다. 그리고, B-트리는 단말 노드들을 순차적으로 접근할 수 있도록 구성되어져 있다. HB-트리는 B-트리에 기반하고 있으므로 트리의 단말 노드를 처음에서 끝까지 순차적으로 접근 가능하다.

HB-트리는 해쉬 조인(hash join)에 응용될 수 있다. 해쉬 조인 기법은 각 릴레이션의 튜플들을 해쉬 함수를 통하여 해쉬값에 따라 정렬된 해쉬 테이블을 생성시키고, 서로간의 해쉬값들을 비교하여 조인을 수행하는 기법이다 [16].

HB-트리의 비단말 노드에는 해쉬키들이 순차적으로 정렬되어 저장되어 있다. 이는 해쉬 조인 기법에서 해쉬키들을 디렉토리에 순차적으로 정렬한 것과 같다. 즉, 단말 노드 하나가 해쉬키값의 범위에 따라 나누어진 디렉토리 분할과 일치한다. 따라서 HB-트리를 인덱스로 갖는 두 릴레이션의 해쉬 조인에 응용할 수 있다. 조인에 참가하는 릴레이션에 대한 해쉬 테이블이 이미 생성되어 있으므로 HB-트리를 처음부터 순차적으로 읽어나가며 해쉬키들을 비교하면 된다.

5 HB-트리의 성능 평가

본 논문에서 제안하고 구현한 HB-트리의 성능을 알아보기 위해 성능 평가를 수행하였다. 실제 구현은 C++ 컴파일러 g++ version 2.7.2을 이용하였고 Solaris 2.5.1하에서 실험되었다. 데이타베이스의 나머지 부분은 서울대학교 OOPSLA연구실에서 만든 SRP(**S**NU **R**elational DBMS **P**latform)을 이용하였다. SRP는 다중 쓰레드를 사용하고 가변 길이의 레코드가 지원되며, 레코드 단위의 잠금을 지원하는 클라이언트-서버형 데이타베이스이다.

SRP에 구현되어 사용되고 있는 B-트리 인덱스와 HB-트리 인덱스의 성능을 비교한다. 성능 평가는 3.2절에서 분류한 연산들을 수행시키고, 수행에 걸린 시간과 트리의 높이, 페이지 접근 회수 등을 비교해 본다.

5.1 성능 평가 모델

먼저 성능 평가에 사용한 고정적인 변수를 살펴보겠다.

입출력의 기본 단위가 되는 페이지의 크기는 $4K$이다. 탐색키의 크기는 문자열 100 바이트로 잡았다. 예를 들어, 사원 릴레이션의 이름이나 주소 애트리뷰트를 고려해 볼 때, 애트리뷰트의 최대 크기는 모든 학생의 이름이나 주소를 저장하기에 적합하도록 충분히 크게 명시되어야 한다. 이름이나 주소의 평균 길이는 작을 수 있더라도 그 최대 길이를 50–100 바이트로 잡는 것은 일반적이라 할 수 있다 [2].

[7] [20]에는 부가의 자료구조를 이용하여 순서가 유지되는 해쉬 스킴을 제안하고 있다.

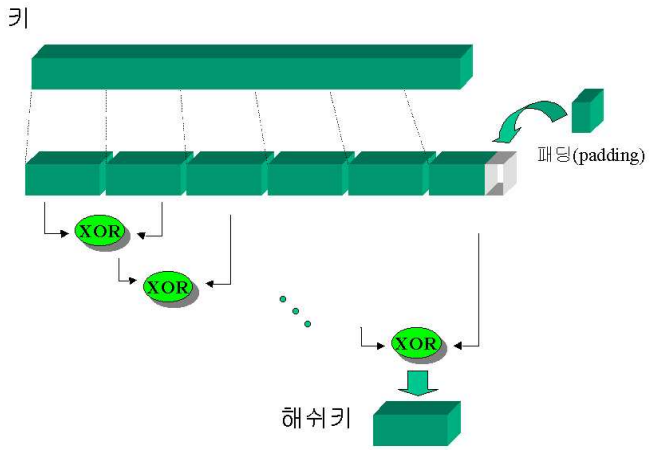

그림 3: 해쉬 함수

해쉬키의 크기는 4 바이트이다. 이는 2^{32} 의 서로 다른 키를 생성해 낼 수 있다. 해쉬키의 크기는 탐색키의 크기와 충돌의 빈도를 고려하여 조정할 수 있다.

탐색키는 다중 필드로 구성된 크기가 큰 문자열로 가정한다. HB-트리는 최종적인 비교를 수행하는 과정이 있으므로 문자열이 아닌 다른 자료형으로 구성된 키도 지원가능하다. 탐색키는 C 언어의 랜덤함수를 호출하여 생성하는데 다음의 두 가지 형태로 만든다.

1. 완전 무작위 생성 : 모든 문자를 랜덤함수를 호출하여 생성한다. 이러한 경우 B-트리에서는 탐색키가 거의 유일한 키값이 생성된다.

2. 부분 무작위 생성 : 문자열의 마지막 10 바이트만을 랜덤함수를 호출하여 생성한다. 즉 $AAAA...AAxxxx$의 형태를 띤다(x : 임의의 문자). 이와 같은 경우 B-트리의 구성시 병목현상을 기대할 수 있다.

해쉬키의 생성 방법은 그림 3과 같다. 이 방법은 다소 충돌이 많이 일어나기는 하나 부가의 자료구조가 필요하지 않아 구현이 간단하므로 채택하였다. 변환되기 전의 키를 잘라 XOR(exclusive-or) 연산을 거듭 수행하여 원하는 크기의 정수값을 생성해 낸다 [13]. 이 때, 키가 최대 키 크기보다 작을 때에는 공백 문자를 끼워 넣음으로써 2.2의 System R에서 발생한 문제가 일어나지 않도록 한다.

HB-트리의 단말 노드 구조는 그림 2의 양식 3을 선택하였다. 참고로 현재 SRP에 구현되어 있는 B-트리의 단말 노드 구조는 양식 2를 따르고 있다.

이러한 단말 노드의 구조에 따라 HB-트리의 삽입 알고리즘을 기술해 보면 다음과 같다.

1. 삽입 레코드의 키에 해쉬 함수를 적용하여 해쉬키를 생성한다.

2. 해쉬키가 삽입될 단말 노드를 찾는다. 이 때, 비단말 노드들은 해쉬키들로 구성되어 있으므로 해쉬키값을 비교하며 탐색을 수행한다.

3. 단말 노드에 삽입할 공간이 충분할 경우

   (1) 단말 노드에 삽입 레코드의 해쉬키와 동일한 해쉬키가 존재할 경우, 해당 슬롯에 키값이 존재하면 [레코드 식별자]만을 첨가시키며 키값이 존재하지 않을 경우 [키, 레코드 식별자]를 저장한다.

   (2) 단말 노드에 삽입 레코드의 해쉬키와 동일한 해쉬키가 존재하지 않을 경우 새로운 슬롯을 할당받아 [해쉬키, 키, 레코드 식별자]를 저장한다.

4. 단말 노드에 삽입할 공간이 충분하지 않을 경우

   (1) 단말 노드를 두 개의 노드로 분할(split)시키고 3과 같은 방식으로 새로운 레코드를 삽입한 후 레코드들을 두 노드에 균등히 저장(node balancing)한다.

   (2) 다음번 탐색을 위해 비단말 노드에 분할된 노드의 첫번째 해쉬키와 페이지 번호를 삽입한다. 이 때, 비단말 노드 또한 분할될 경우 이들의 분할을 지속적으로 상위 노드들로 전파시키며, 만일 루트(root) 노드가 분할할 경우 트리의 높이가 하나 증가하게 된다.

HB-트리의 삭제 알고리즘은 다음과 같다.

1. 삭제 레코드의 키에 해쉬 함수를 적용하여 해쉬키를 생성한다.

2. 해쉬키가 삭제될 단말 노드를 찾는다.

3. 단말 노드에 삭제할 공간이 충분할 경우

   (1) 해당 슬롯에 키값이 하나 존재하면 [해쉬키, 키, 레코드 식별자]를 모두 삭제한다.

   (2) 해당 슬롯에 키값이 하나 이상 존재하면 [키, 레코드 식별자]를 삭제한다.

4. 키를 삭제함으로써 이웃 노드(sibling node)와 병합(node merging)할 수 있을 경우 두 노드의 레코드를 하나의 노드로 합하여 저장한다.

5. 다음번 탐색을 위해 비단말 노드에 병합된 노드의 첫번째 해쉬키와 페이지 번호를 삽입한다. 이 때, 비단말 노드 또한 병합될 경우 이들의 병합을 지속적으로 상위 노드들로 전파시키며, 만일 루트(root) 노드가 병합할 경우 트리의 높이가 하나 감소하게 된다.

5.2 성능 평가 결과

5.2.1 부모 릴레이션에의 삽입 연산

그림 4의 그래프는 각각 B-트리와 HB-트리를 인덱스로 갖고 있는 부모 릴레이션에 $100,000$개의 레코드를 삽입시면서 얻은 결과이다. 각각의 데이타는 같은 실험을 5번 수행하여 평균값을 구한 것이다.

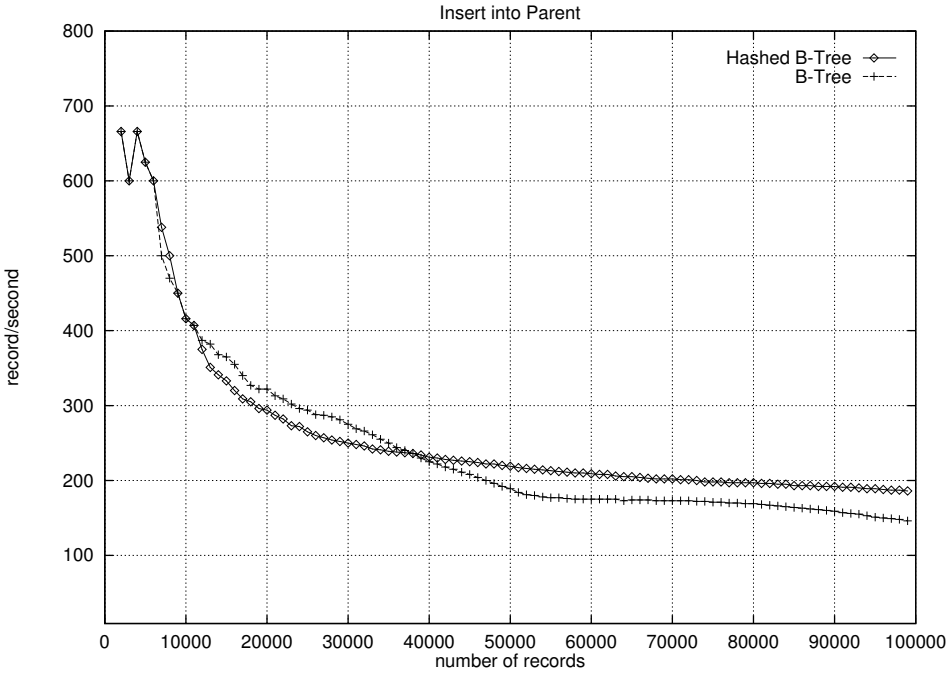

그림 4: 부모 릴레이션에의 삽입 연산 결과

x축은 트리에 총 삽입된 레코드의 개수를 , y축은 x개의 레코드를 삽입시켰을 때 걸린 시간을 측정하여, 단위 시간(초)당 삽입된 레코드의 개수를 표시하였다.

부모 릴레이션에 대한 삽입 연산은 먼저 삽입될 레코드가 기본 키 제약을 만족하는지 조사한다(3.2.1절 참조). 삽입될 키가 이미 트리에 존재하는지 탐색해 보고 트리에 존재하면 이는 기본 키 제약 조건을 만족치 않으므로 삽입 연산을 수행하지 않고, 존재치 않으면 삽입한다.

레코드 삽입에 따라 트리의 높이가 변하는 구간은 다음과 같이 조사되었다.

트리 높이	B-트리	HB-트리
3	1,000개 이전	6,000개
4	27,000개	100,000 이후

HB-트리에서 해쉬키 생성시 충돌은 약 4.5%로 나타났다. B-트리와 HB-트리의 높이가 모두 3인 30,000개 이전 구간에서는 B-트리의 성능이 약간 더 좋게 나오는데, 이는 B-트리의 키는 충돌이 전혀 발생되지 않도록 생성된 반면 HB-트리는 충돌이 발생하여 그에 따르는 부가의 비교 연산이 발생하기 때문이다. B-트리의 높이가 4로 증가하는 30,000 개 이후 구간부터는 B-트리의 삽입율이 감소하여 HB-트리의 성능이 더 좋아짐을 살펴볼 수 있다.

5.2.2 자식 릴레이션에의 삽입 연산

부모 릴레이션에 50,000개의 레코드를 먼저 삽입시켜 두고, 자식 릴레에션에 50,000개의 레코드를 참조 무결성 제약 조건을 만족시키며 삽입시킨다. 자식 릴레이션에 삽입될 키는 부모 릴레이션

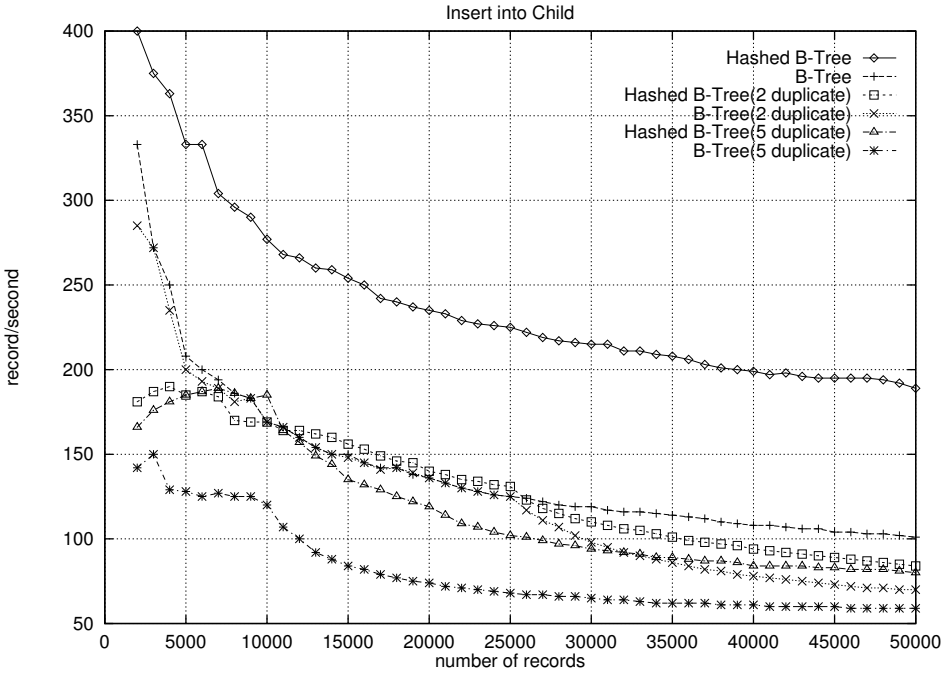

그림 5: 자식 릴레이션에의 삽입 연산 결과

의 외래 키가 되므로 우선 부모 릴레이션의 기본 키로 이 키가 존재하는지 탐색한 후 자식 릴레이션에 삽입한다(3.2.4절 참조).

그림 5의 그래프는 위 삽입 연산의 수행 결과를 나타낸다. 자식 릴레이션에 키를 삽입할 때는 중복키가 없는 경우, 중복키가 둘씩 있는 경우, 중복키가 다섯개씩 있는 경우의 성능을 각각 조사하였다. 키가 중복되어 있을 경우, B-트리에서도 해당 튜플을 찾기 위해 레코드 아이디들을 비교해야 하는 필요가 생긴다[8]. $50,000$개의 키를 무작위로 생성할 때 중복되는 키가 생기지 않으므로 임의로 중복되는 키를 생성하여 삽입시켜 보았는데, HB-트리가 중복키가 생김에 따른 성능의 저하가 B-트리보다 적음을 알 수 있다.

또한, 부모 릴레이션에의 삽입 연산에서보다 HB-트리의 성능이 B-트리보다 훨씬 좋음을 확인할 수 있는데, 이는 외래 키 조건의 만족 여부를 알아보기 위해 부모 릴레이션을 검색할 경우, 부모 릴레이션의 HB-트리는 높이가 3인 반면, B-트리는 높이가 4이므로 훨씬 많은 페이지 접근이 발생되기 때문이다.

페이지 접근 회수는 그림 6에서 확인할 수 있다. 이 그래프는 자식 릴레이션에 삽입 연산을 수행하면서 실제 디스크에서 데이타베이스의 버퍼로 올라오는 페이지의 개수를 조사한 것이다. $1,000$개의 레코드를 삽입할 때마다 발생하는 페이지 접근 회수를 y축에 표시하였다. 그래프를 보면, B-트리에서의 페이지 접근 회수가 HB-트리보다 많음을 알 수 있고, B-트리의 높이가 변하는 $30,000$개 부근에서 페이지 접근 회수가 계단 모양으로 증가하는 것을 볼 수 있다.

그림 7은 그림 5와는 달리 $1,000$개의 레코드가 삽입될 때마다 소요되는 시간을 측정하여 이를

[8]SRP의 B-트리에서 같은 키를 갖는 레코드 아이디들은 그 값에 따라 정렬되어 있다.

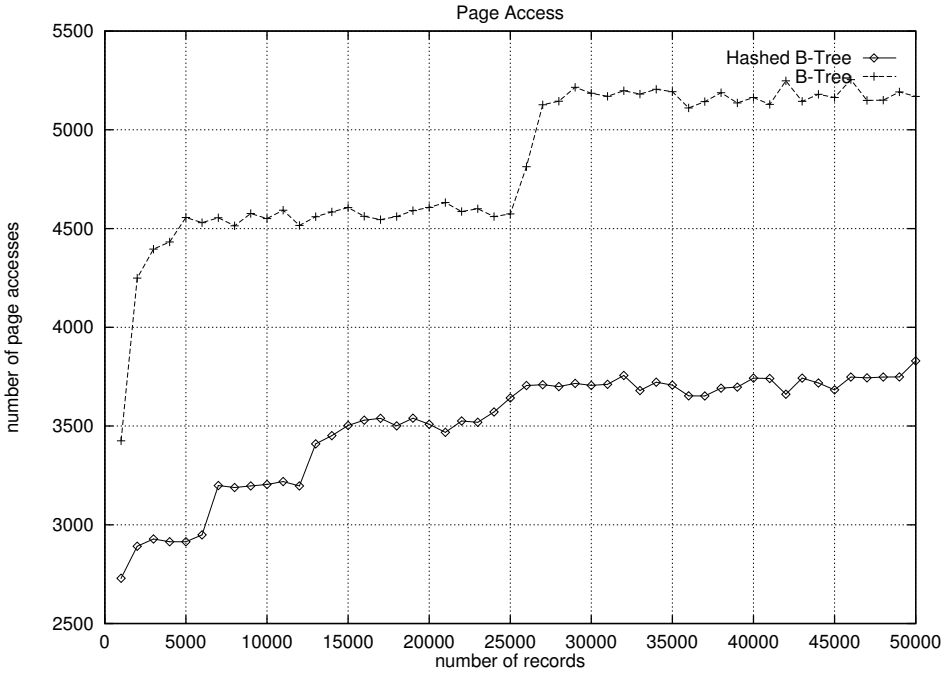

그림 6: 페이지 접근 회수

초당 삽입된 레코드의 개수로 나타낸 그래프이다. 트리의 높이가 변함에 따라 계단형의 모양을 보임을 알 수 있다.

그림 8은 키를 완전 무작위로 생성하였을 때와 부분 무작위로 생성하였을 때의 성능을 비교한 그래프이다. 사원 릴레이션의 이름 애트리뷰트와 같은 경우, 성이 같은 사원들의 데이타는 트리의 동일 페이지에 밀집되어 있을 가능성이 크다. 그림에서 보면 HB-트리는 원래 키값을 해싱을 통해 재분산시킴으로써 B-트리에서 발생하는 병목 현상을 줄여줌을 알 수 있다. 이는 인덱스를 여러 트랜잭션이 병행적으로 접근할 때 병행성 제어의 성능을 높여줄 수 있음을 말해준다.

5.2.3 자식 릴레이션에의 갱신 연산

부모 릴레이션에 50,000개의 레코드, 자식 릴레이션에 30,000개의 레코드가 삽입되어 있을 때, 자식 릴레이션에 5,000개의 레코드를 갱신한다. 자식 릴레이션에 삽입될 키는 부모 릴레이션의 외래키가 되므로 우선 부모 릴레이션의 기본 키로 이 키가 존재하는지 탐색한다. 참조 무결성이 만족되면 이전의 키를 삭제하고 갱신될 키를 삽입한다(3.2.6절 참조).

그림 9는 500개의 레코드가 갱신될 때마다 소요되는 시간을 측정하여 초당 갱신된 레코드의 개수를 조사하여 나타낸 그래프이다. 구간에 대해 일정한 성능을 나타내고 있다.

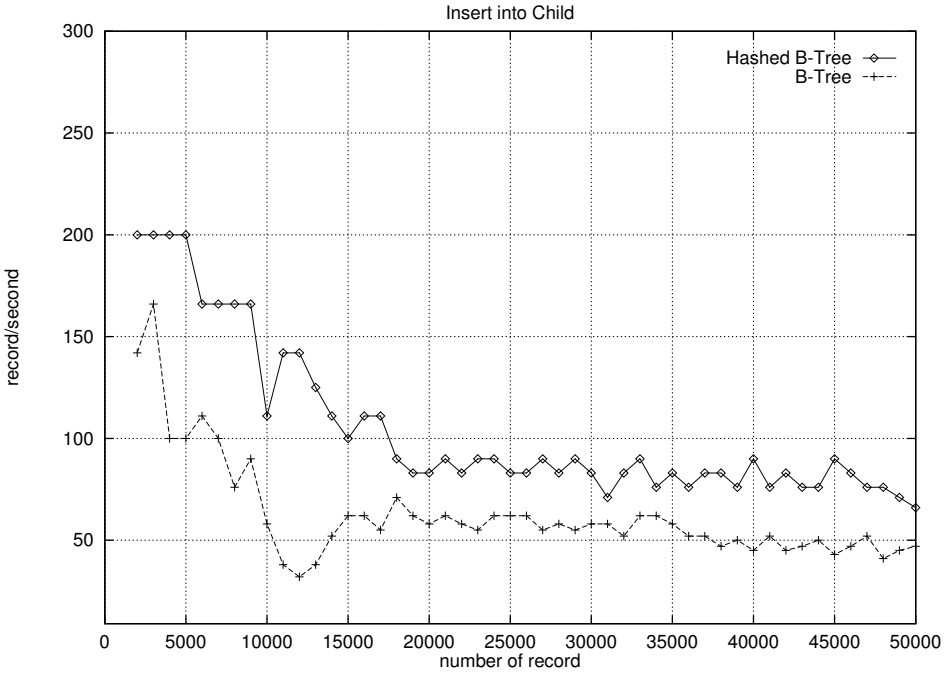

그림 7: 자식 릴레이션에의 삽입 연산 결과

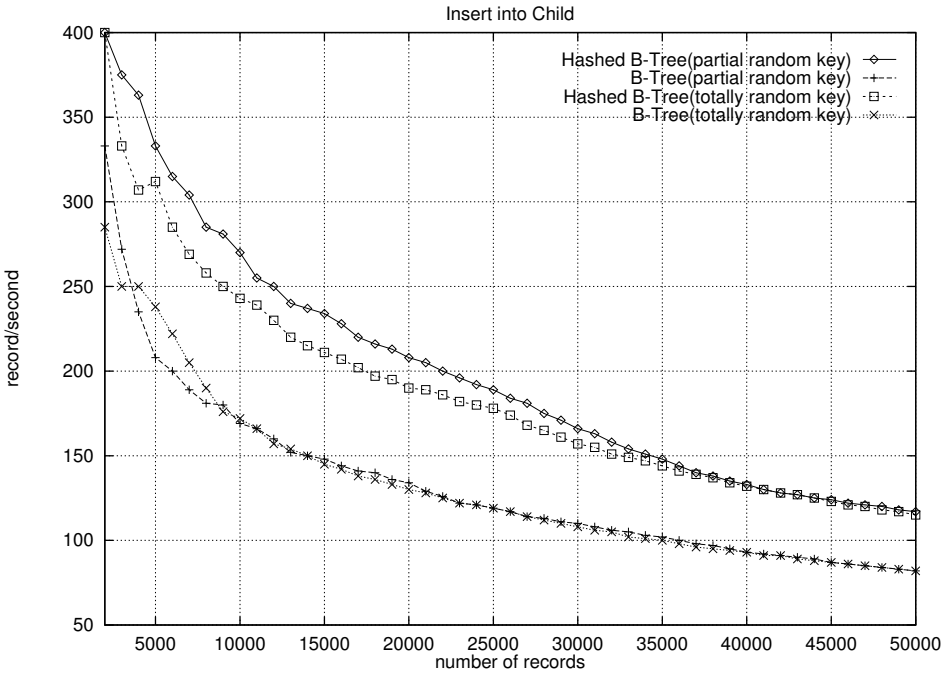

그림 8: 자식 릴레이션에의 삽입 연산 결과

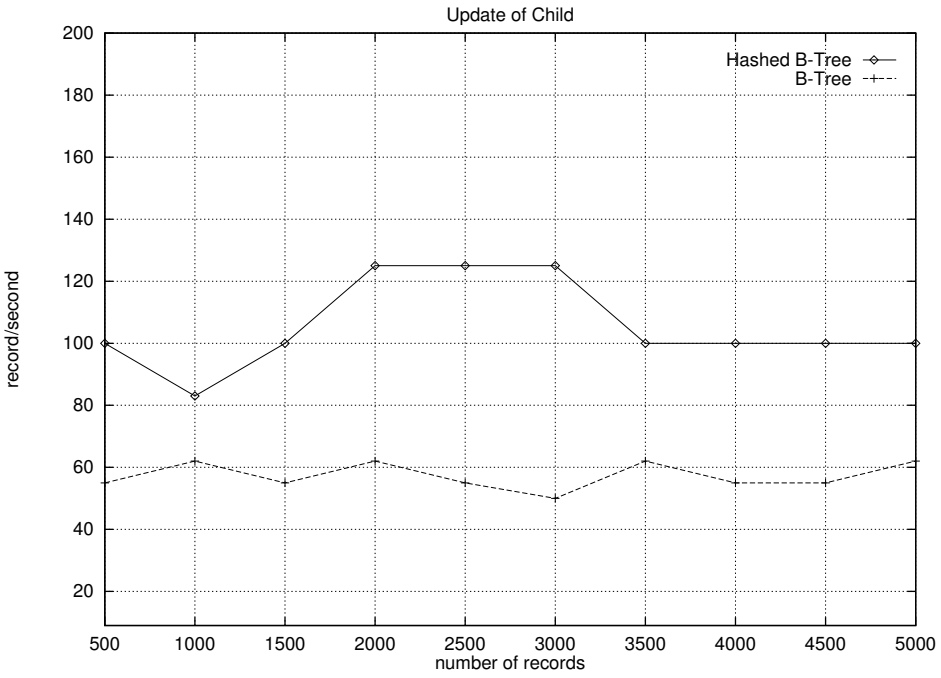

그림 9: 자식 릴레이션에의 갱신 연산 결과

5.2.4 부모 릴레이션에의 갱신 연산

부모 릴레이션에 50,000개의 레코드, 자식 릴레이션에 30,000개의 레코드가 삽입되어 있을 때, 부모 릴레이션에 3,000개의 레코드를 갱신한다. 갱신될 키는 기본 키 제약 조건을 만족하여야 하므로, 부모 릴레이션에 이미 갱신될 키가 존재하는지를 조사한다. 만약 존재하지 않는다면 갱신 이전의 키를 트리에서 삭제하고, 갱신될 키를 삽입한다. 갱신 이전의 키를 참조하고 있던 자식 릴레이션의 튜플들은 부모 릴레이션을 검색하여 참조하던 키가 갱신 이후에도 존재하는지에 대한 여부를 결정한다. 이 때, 참조할 키가 존재치 않고 참조 무결성이 SET DEFAULT 옵션으로 제약되고 있을 경우, 자식 릴레이션의 키들은 미리 정해둔 디폴트 키로 바꾸어야 한다. 디폴트 키가 부모 릴레이션에 존재하는지 먼저 검사한 후 자식 릴레이션에 대한 삭제와 삽입 연산을 수행한다(3.2.3절 참조).

그림 10은 250개의 레코드가 갱신될 때마다 소요되는 시간을 측정하여 초당 갱신된 레코드의 개수를 조사하여 나타낸 그래프이다. 복잡한 연산을 수행하므로 자식 릴레이션에의 갱신 연산보다 성능이 저하됨을 알 수 있다. 갱신 개수가 늘어남에 따라 구간별 성능이 HB-트리에서 급격히 저하되는 이유는 디폴트 키를 갖는 레코드가 늘어남에 따라 오버플로우 페이지의 관리 부하가 B-트리에 비해 증가하기 때문이다.

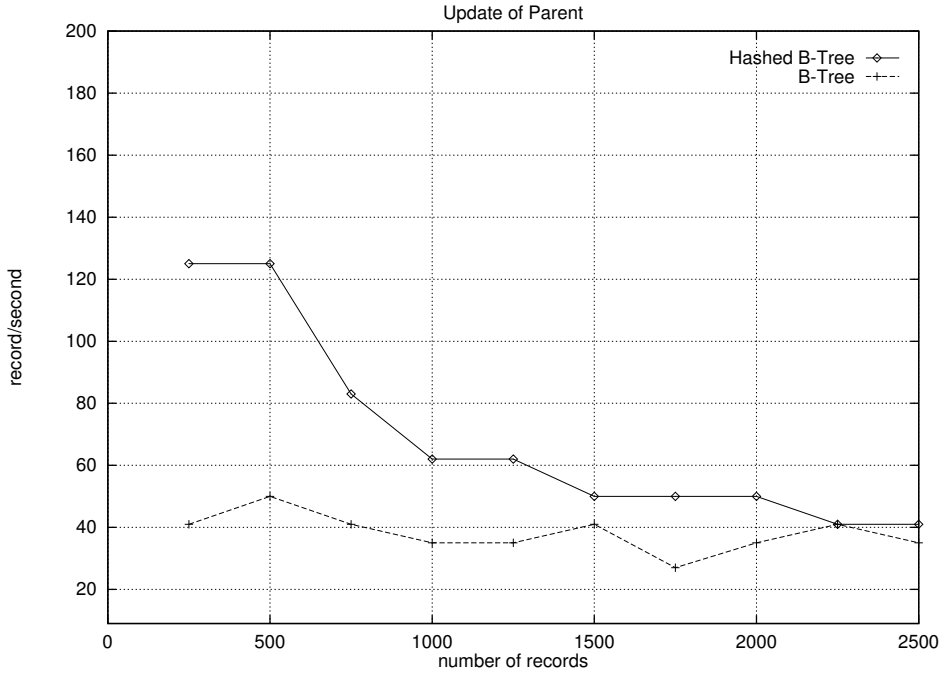

그림 10: 부모 릴레이션에의 갱신 연산 결과

6 결론

본 논문에서는 무결성 제약을 시스템에서 지원하기 위해 필요한 다량의 데이타 접근을 효율적으로 처리하기 위한 인덱싱 기법에 대하여 이야기하였다. 무결성 제약을 만족시키기 위해 필요한 질의 패턴들이 종점 질의로 구성되어 있다는 점에 착안하여, 논문에서 제안한 접근 경로인 HB-트리는 B-트리에 키를 해싱을 통해 압축하여 저장한다. 이로써 접근 경로에의 탐색 시간을 줄이고, 데이타 저장의 효율을 꾀하였다.

여러 트랜잭션이 병행적으로 데이타베이스를 접근하는 경우 인덱스에 대한 병행성 제어와 고장 발생시의 회복 기법이 필요하다. B-트리에 대한 효율적인 병행성 제어와 회복 기법은 이미 많은 시스템에서 구현되어져 있는 반면 [17], 해싱 기법에 대한 효과적인 병행성 제어와 회복 기법은 B-트리에 비해 매우 적다. HB-트리는 구현시 기존의 시스템에 많은 수정을 요하지 않고, B-트리의 병행성 제어나 회복 기법을 그대로 사용할 수 있는 장점이 있다.

참고문헌

[1] 이상구, 장재영. “무결성 제약의 효율적 검증을 위한 최적화 방법”. **정보과학회논문지*(B)***, 제 23 권 제 5 호:466–476, 1996년 5월.

[2] G. Antoshenkov, D. Lomet, and J. Murray. “Order Preserving String Compression”. *International Conference on Data Engineering(ICDE)*, pages 655–663, 1996.

[3] D. Z. Badal, G. J., and Popek. "Cost and Performance Analysis of Semantic Integrity Validation Methods". *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 109–115, May 1979.

[4] R. Bayer and K. Unterauer. "Prefix B-Trees". *ACM Trans. Database Syst.*, 2(1):11–26, Mar. 1977.

[5] D. A. Bell and S. M. Deen. "Key Space Compression and Hashing in PRECI". *The Computer Journal*, 25(4), 1982.

[6] D. A. Bell and S. M. Deen. "Hash Trees Versus B-Trees". *The Computer Journal*, 27(3), 1984.

[7] M. Blasgen, R. Casey, and K. Eswaran. "An Encoding Method for Multi-field Sorting and Indexing". *Communications of ACM*, 20(11):846–876, Nov. 1977.

[8] C. J. Date with Hugh Darwen. *A Guide to The SQL Standard.* Addison Wesley, 1993.

[9] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of ACM*, 13:377–387, 1970.

[10] C. J. Date. "Referential Integrity". *Proc. of the Conf. on VLDB*, pages 2–12, Sept. 1981.

[11] S. M. Deen. "An Implementation of Impure Surrogates". *Proc. of the Conf. on VLDB*, pages 245–246, 1982.

[12] K. P. Eswaran and D. D. Chamberlin. "Functional Specification of a Subsystem for Data Base Integrity". *Proc. of the Conf. on VLDB*, Sept. 1975.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman Publishers, Inc., 1993.

[14] T. Härder and J. Reinert. "Access Path Support for Referential Integrity in SQL2". *VLDB Journal*, 5(3):196–214, July 1996.

[15] J. Melton and A. R. Simon. *Understanding the new SQL : A Complete Guide.* Morgan Kaufman Publishers, Inc., 1993.

[16] P. Mishra and M. H. Eich. "Join Processing in Relational Databases". *ACM Computing Surveys*, 24(1), 1992.

[17] C. Mohan. "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging". Technical Report RJ6846, IBM Research Division, Aug. 1989.

[18] C. Mohan. "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transaction Operating on B-Tree Indexes". In *Proc. of the Conf. on VLDB*, pages 392–405, Brisbane, Australia, Aug. 1990.

[19] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging". *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[20] E. J. Otoo. "Linearizing the Directory Growth in Order Preserving Extendible Hashing". *International Conference on Data Engineering(ICDE)*, pages 580–588, 1988.