관계형 데이타베이스를 위한 응용 프로그램 독립적인 스키마 진화

(Application Program Independent Schema Evolution in Relational Databases)

나 영 국 †

(Young-Gook Ra)

요 약 데이타베이스 스키마는 모델링 되는 환경이 변화할 때에도 여전히 유효한 상태로 남을 만큼 안정적이라고 가정되어 왔다. 그러나 실제로는, 데이타 모델은 데이타베이스 설계자들이 공통으로 가정하는 만큼 안정적이지 않다. 현재 데이타베이스 시스템에서 풍부한 스키마 변화 연산들이 제공되지만은 사용자들은 스키마 변화가 스키마에 쓰여진 기존의 응용 프로그램에 영향을 미치는 문제로 곤란을 겪어왔다. 이 논문은 응용 프로그램에 영향을 주는 문제를 탐구한다. 옛 스키마에 기존의 프로그램을 지속적으로 지원하기 위하여 옛 스키마가 이전처럼 변경과 질의를 계속 허락해야 한다. 더 나아가, 관련 데이타는 최신 상태로 유지되어야 한다. 이것을 스키마 변화 도구의 프로그램 독립 성질이라 부른다. 이 성질을 달성하기 위하여, 이 논문은 프로그램 독립적인 스키마 진화 (Program Independency Schema Evolution: PISE) 방법론을 제안한다. 더 나아가, 각각의 관계형 스키마의 변화 연산들에 대하여 PISE 방법론에 기초한 구현 알고리즘을 도식적으로 설명함으로써 PISE 방법론의 포괄성과 견고성을 증명한다.

키워드 : 스키마 진화, 프로그램 독립성, 데이타베이스 뷰, 관계형 데이타베이스, 데이타 모델, 스키마 버전, 데이타베이스 스키마, 용량-증가 스키마 변화, 타입 불일치, 공유 데이타베이스

Abstract The database schema is assumed to be stable enough to remain valid even as the modeled environment changes. However, in practice, data models are not nearly as stable as commonly assumed by the database designers. Even though a rich set of schema change operations is provided in current database systems, the users suffer from the problem that schema change usually impacts existing application programs that have been written against the schema. In this paper, we are exploring the possible solutions to overcome this problem of impacts on the application programs. We believe that for continued support of the existing programs on the old schema, the old schema should continue to allow updates and queries, as before. Furthermore, its associated data has to be kept up-to-date. We call this the program independency property of schema change tools. For this property, we devise so-called program independency schema evolution (PISE) methodology. For each of the set of schema change operations in the relational schemas, the sketch of the additional algorithms due to the PISE compliance is presented in order to prove the comprehensiveness and soundness of our PISE methodology.

Key words : schema evolution, program independency, database view, relational database, data model, schema version, database schema, capacity-augmenting schema change, type mismatch, shared database

1. 서 론

데이타베이스 설계자는 데이타베이스 시스템에 모델링 되는 환경을 정확히 반영하려는 목적으로 스키마를 생성한다. 결과 스키마는 모델 되는 환경이 변화할 때도 유효하게 남을 정도로 안정적이라 가정한다. 그러나 실제에 있어서는 데이타 모델은 데이타베이스 설계자가 통상적으로 가정하는 만큼 안정적이지 못하다. 스키마 변화는 릴레이션 혹은 속성의 단순한 더하기 빼기 뿐만 아니라 복잡한 릴레이션을 몇 개의 단순한 릴레이션

· 본 연구는 2003년에 서울시립대의 지원을 받았음
† 정 회 원 : 서울시립대학교 전자전기컴퓨터공학과 교수
ygra@uos.ac.kr
논문접수 : 2004년 1월 30일
심사완료 : 2004년 7월 7일

으로 나누는 등의 연산을 포함한다. 스키마 변화가 전문화된 도구에 의해 다루어진다면, 특히 도구의 기능이 무결성 확인과 옛 스키마에 따른 저장된 데이타베이스를 새 스키마의 구조에 맞게 변환하는 기능을 포함한다면, 데이타베이스 설계자의 일을 많이 덜어 줄 것이다. 실제로 많은 그러한 도구와 방법론이 제안되었다[1-10].

현재 데이타베이스 시스템에서 풍부한 스키마 변화 연산들이 제공된다 하여도 사용자는 스키마에 쓰여진 기존의 응용 프로그램에 스키마 변화가 영향을 끼치는 문제로 곤란을 겪어왔다. 그래서 현재 DBMS에서 스키마 변경 능력은 지원되는 스키마 변화 언어 보다는 기존 프로그램에 미치는 영향에 더욱 제한된다. 이 문제는 스키마 버전 시스템을 이용하여 부분적으로 해결될 수 있다[11]. 왜냐하면 그들은 새 프로그램이 원하는 새 스키마에서 개발될 때 옛 스키마 버전을 유지하여 여전히 옛 버전에 기존의 프로그램이 동작하도록 허락하기 때문이다. 그러나 스키마 버전 시스템에서 옛 스키마는 단지 읽기 모드에서만 접근 가능하다. 이는 옛 프로그램이 그들의 스키마를 통해서 낡은 데이타에 접근하는 것을 의미한다.

옛 스키마에 동작하는 기존 프로그램의 지속적인 운영을 위해서 스키마 변경 후에도 계속하여 옛 스키마에 대한 변경과 질의가 허락되어야 한다. 더 나아가 관련 데이타는 최신 상태로 유지되어야 한다. 이를 스키마 변화의 프로그램 독립 성질(PISE: Program Independent Schema Evolution)이라 부른다. 이 논문에서 정의하는 프로그램 독립 성질은 어떤 스키마 버전(VS1)의 주키 필드 값들은 그들이 다른 어떤 스키마 버전(VS2)을 통하여 생성되었는지에 관계없이 VS1 버전에서 접근 가능하여야 한다는 것이다. 비-키 속성 (a) 값들은 모든 버전을 통해서 그들을 결정하는 주키 속성 (b) 값들부터 유일하게 결정될 수 있으면 VS1에서 접근 가능하며 그렇지 못하고 버전 내 또는 버전 간 불일치가 있으면 VS1에서 *a* 속성 값은 *null*로 지정된다. 더욱이 VS1에서 접근 가능한 키 속성 값들에 대한 다른 버전 VS2에서의 변경은 VS1에 전파되고 비-키 속성 값들에 대한 변경은 그 변경이 버전 내 또는 버전 간 불일치만 초래하지 않으면 VS1에 전파된다.

스키마 버전에서 기존 프로그램의 지속적인 운영을 위한 프로그램 독립 성질의 중요성을 인식하고 해결책을 제시한 연구들이 있다[12-14]. 기본적으로 그들은 다른 버전과의 타입 불일치를 해소하는데 프로시저를 사용한다. 그들은 각 스키마 변화마다 사용자가 제공하는 프로시저를 요구한다. 하지만 사용자가 스키마가 변할 때마다 타입 불일치의 모든 경우를 조사하고 각 경우마다 적절한 프로시저를 제공할 것을 기대하는 것은 비합리적이다.

뷰 접근 방식은 타입 불일치 문제의 해결을 위한 다른 접근 방식으로 여러 연구자들에[15,16] 의해 제안 되었다. 이 접근 방식의 기본 원리는 데이타베이스 뷰가 특정 스키마 버전에 해당하는 인스턴스들의 한 시각을 제공하게 한다는데 있다. 뷰 접근 방식은 뷰가 다른 시각을 제공하고 뷰에 의해 유도된 시각에 대한 갱신이 잘 연구되어 있고 유도된 시각을 발생하는 기능 즉, 질의를 쉽게 최적화할 수 있다는 데서 프로시저 접근 방식보다 우수하다고 판단된다.

그러나 Tresch와 Scholl[16]은 뷰 기반 접근 방식은 스키마 진화 문헌[1]에서 제안된 스키마 진화의 전 범위를 지원하지 않는다는 것을 발견하였다. 특히, 저장되는 이름 붙여진 질의[17]로 정의된 뷰 메커니즘은 기저의 데이타베이스의 유도된 함수라는 태생의 제약에 기인하여 용량-증가 스키마 변화를 달성할 수 없다. Bertino [15]와 이전 연구[18]는 용량-증가 변화가 뷰를 이용하여 달성될 수 있도록 추가적인 스키마 요소를 더할 수 있게 뷰를 확장하는 것을 제안하였다. 이전 연구[18]에서 실제로 그러한 뷰 메커니즘을 구축하였지만 벤치마크를 통한 성능 실험에서 용량-증가 뷰가 효율적인 방법으로 구현될 수 없다는 것을 보여주었다. 이 논문은 실제적인 해결책을 제안한다. 그리고 접근 방식은 비-용량-증가 변화에 국한되지도 않고 용량-증가 뷰 메커니즘도 요구하지 않는다. 본 논문이 주장하는 해결책은 대부분의 관계형 DBMS에서 제공되는 표준적인 스키마 진화 도구와 뷰 메커니즘만 요구한다. 이 대안의 해결책은 프로그램 독립적인 스키마 진화(PISE) 방법론이라 불리우며 다음의 간단한 아이디어에 기반한다. PISE 방법론은 비-용량-증가 스키마 변화 연산을 위해서는 연산이 의도하는 뷰 스키마를 유도하고 용량-증가 스키마 변화 연산을 위해서는 (1) 스키마를 연산에 의해 요구되는 추가의 용량을 위해 직접적으로 변경하고 (2) 목표 스키마가 변경된 스키마에 기반한 뷰로써 발생하고 (3) 원래의 스키마는 변경된 스키마로부터의 뷰로써 재생된다.

다음 절은 본 논문에서 제시하는 접근 방식을 개괄적으로 설명한다. 3절은 스키마 진화 연산자들을 연산자의 프로그램 독립성을 달성하는 알고리즘의 도식적인 설명과 함께 소개한다. 4절은 관련 연구를 논의한다. 마지막 5절은 미래 연구와 함께 결론을 낸다.

2. PISE 방법론의 개관

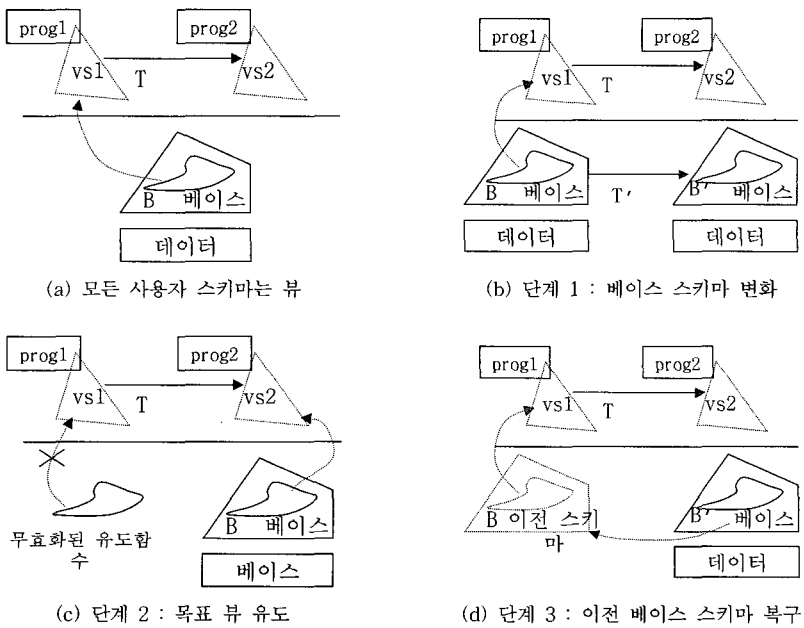
PISE 방법론에서 모든 스키마는 - 사람 사용자가 직접적으로 인터액티브하게 또는 응용 프로그램을 통하여 접근하는 - 그림 1(a)에서 보여지듯이 실제적으로는 하나의 기저 베이스 스키마로부터의 뷰들이다. 기존 프로그램 Prog1이 VS1(View Schema 1) 스키마에 동작하고 새롭게 개발되는 프로그램 Prog2가 새로운 데이타베

이스 스키마를 요구한다고 한다. 새 데이타베이스 스키마를 위하여 데이타베이스 관리자는 VS1을 목표 스키마 VS2로 변환할 목적으로 VS1에 스키마 변환, T를 지정한다. 그러면 그림 1(a)에서 보여지듯이 원하는 목표 스키마가 새로운 버전 번호와 함께 뷰 VS2로써 생성된다. 용량-증가와 비-용량-증가 경우와 마찬가지로 용량-증가의 경우에도 뷰로써 원하는 스키마를 발생시키는 PISE 방법론은 세 단계로 구성된다. 각 단계는 다음에서 상세하게 설명한다.

처음에 원하는 목표 스키마를 발생하는데 필요한 용량만큼 기저 베이스 스키마 B를 증가시킨다. T가 비-용량-증가인 경우 이 단계는 생략된다. 이는 베이스 스키마 변화 단계라 불리운다. 그림 1(b)에서 베이스 스키마 B는 이 단계에 의해 B′로 직접 변화한다. 즉, B가 사라진다. 그러면 PISE 방법론은 그림 1(c)에서 보여지듯이 증가된 베이스 스키마 B′에 뷰 유도를 적용하여 원하는 타겟 스키마(VS2)를 발생한다. 이 단계는 목표 뷰 유도 단계라 불리운다. 사용자 스키마 VS1은 이전 단계에서 변경되어 더이상 유효하지 않게 될 수도 있음을 주의하라. 마지막으로 그림 1(d)에서 보듯이 원래 베이스 스키마 B가 변경된 베이스 스키마 B′에 정의된 뷰로써 재생성된다. 이 단계는 옛 사용자 스키마 VS1의 원천 스키마 즉, 원래의 베이스 스키마 B를 복구한다. 그러므로 VS1은 다시 유효하게 된다. 이는 이전 베이스 스키마 복구 단계라 불리운다. 이 단계는 T가 비-용량-증가의 경우 베이스 스키마가 변경되지 않았기 때문에 생략된다. 그림 1(d)에서 옛 프로그램 Prog1이 스키마 변환, T에도 불구하고 같은 스키마 VS1에 계속 동작하는 반면에 Prog2는 원하는 새 스키마 버전 VS2에 대해 이제 개발될 수 있음을 알 수 있다.

PISE(Program Independent Schema Evolution)의 방법론에 대하여 다음을 관찰할 수 있다.

1. 첫째, 스키마(PISE에서 뷰로서 구현된)의 범위는 이 특정 스키마에서 생성된 튜플에 국한되지 않는다. 대신에, 데이타베이스의 모든 튜플은 그 튜플이 속한 릴레이션이 스키마에 포함되기만 하면 그 스키마를 통해 접근 변경될 수 있다. 이는 튜플들은 생성된 특정 스키마 버전에 속하고 옛 스키마는 단지 질의 모드에서만 동작하는 통상적인 스키마 버전 시스템과 대조를 이룬다[3,19]. 이는 튜플 범위 원칙이라 불린다.
2. 데이타베이스의 모든 튜플은 하나의 기저 베이스 스키마에 관련되어 있고 모든 다른 스키마는 단지 그 베이스 스키마에 정의된 뷰이다. 그래서 모든 튜플이 비록 여러 스키마 버전으로부터 접근할 수 있다 하여도 그 튜플들을 위해 단지 하나의 물리적 복사본(그들의 속성값 등)을 가진다. 이는 각 버전에 인스턴스들이 중복되는 버전 시스템과 대조된다. 이는 하나의 원천 원칙이라 불리운다.
3. 어떤 스키마(VS2)의 모든 튜플 값들에 대한 변경은 그 값들이 다른 스키마(VS1)의 주키 속성 값이면 그



(a) 모든 사용자 스키마는 뷰

(b) 단계 1 : 베이스 스키마 변화

(c) 단계 2 : 목표 뷰 유도

(d) 단계 3 : 이전 베이스 스키마 복구

그림 1 PISE 방법론의 기본 가정과 프로그램 독립성을 달성하는 3 단계

스키마에서(VS1) 관찰되고 그 값들이 VS1의 비-키 속성 값이면 VS2에서의 변경이 스키마 내 또는 스키마 간 불일치를 초래하지 않는다면 VS1에서 보인다. 비-키 속성 값이 키 속성 값들로부터 유일하게 결정되지 않으면 VS1에서 그 속성 값은 null로 지정된다. 이 변경 전파 원칙을 프로그램 독립성의 원칙이라 불리운다.

4. 모든 스키마 변화 연산은 기존의 응용 프로그램을 지속적으로 지원하기 위하여 그것이 변경하는 원래 스키마를 보존하여야 한다. 이는 보존 원칙이라 불리운다.
5. 어떤 스키마에 행해진 모든 스키마 변화 연산은 같은 데이타베이스에 존재하는 다른 뷰 스키마에 영향을 주어서는 안 된다. 이는 투명성 원칙이라 불리운다.

3. PISE를 사용한 스키마 변화 연산들

본 논문은 [1]에서 제안된 포괄적인 스키마 변화 연산자 집합을 PISE 방법론으로 구현한다. 이들은 추출(extract), 종속성-도입(import-dependency), 이름-변화(change-name), 속성-더하기(add-attribute), 속성-지우기(delete-attribute), 키-상승(promote-key), 키-강등(demote-key), 분해(decompose), 결합(compose), 나누기(partition), 합병(merge), 도출(export), 도입(import) 등이다. 연산자들 중에서, 속성-더하기, 키-상승, 분해, 도출, 추출 연산자들은 용량-증가이고 나머지는 비-용량-증가이다. 이 절에서 연산자들의 문법(syntax)와 의미(semantics)를 소개한다. 연산자가 용량-증가이면, 원천 베이스 스키마를 변화하고, 목표 뷰를 유도하고, 원래 베이스 스키마를 복원하는 알고리즘이 그림과 함께 설명된다. 추출 연산의 경우는 생성하는 스키마가 변경 연산 대하여 베이스 스키마와 같은 행동을 보인다는 것과 PISE 구현이 프로그램 독립성의 원칙을 만족시킨다는 것을 증명한다.

3.1 이행 종속성의 추출(extract)

이 연산의 문법은 "*추출(rel, R(a_1, a_2, ..., a_m, b_1, b_2, ..., b_n))*" 이다. 의미는 릴레이션 *rel*의 비-키 속성 a_1, a_2, ..., a_m과 b_1, b_2, ..., b_n을 추출하여 키 속성이 a_1, a_2, ..., a_m인 새 릴레이션 *R*을 구성하는 것이다. 릴레이션 *rel*은 속성 b_1, b_2, ..., b_n이 제거되어 작아진다.

[정리 1] 추출 연산은 용량-증가이다.

[증명] 이 연산에서 릴레이션 *rel*이 두개의 릴레이션 *newRel*과 *R*로 나누어지고 릴레이션 *rel*에 속하지 않은 두개 릴레이션 *newRel*과 *R* 어느 하나에 속하는 인스턴스들이 있을 수 있다. 그러한 인스턴스들은 *newRel* 릴레이션의 인스턴스들과 a_1, a_2, ..., a_m 속성 값들을 공유하지 않는 *R*의 인스턴스들일 수 있고 반대로 *R*의 인스턴스들과 a_1, a_2, ..., a_m 속성 값들을 공유하지 않는 *newRel* 인스턴스들 일 수 있다.

[PISE 구현] PISE 방법론에서는 모든 릴레이션은 원천 릴레이션으로부터의 뷰로써 정의된다. 예를 들어 그림 2에서 보듯이 피연산 릴레이션 *rel*은 초기에는 릴레이션 *rel'*으로부터 단지 *k, a, b* 속성들만 프로젝트(project)하는 뷰로써 정의된다. 1 단계에서 원천 릴레이션은 용량-증가된 스키마 요소를 생성하기 위해 물리적으로 변화한다. 예를 들어 원천 릴레이션 *rel'*은 그림 2의 아래 부분에서 보여 지듯이 *newRel'*과 *R'* 릴레이션으로 분해 된다. 2 단계에서 목표 릴레이션은 원천 릴레이션으로부터 유도된다. 예를 들어 목표 릴레이션 *newRel*과 *R*은 그림 2의 오른쪽에 각각 보여지듯이 "select k, a from newRel'" and "select a, b from R'"로 정의된다. 3 단계에서 변화 이전의 원천 릴레이션이 뷰 유도 함수를 이용하여 복구된다. 그림 2의 아래의 점선 블록 화살표는 이 단계를 추출 함수에서 구현한 것을 나타낸다. 이 예에서 복구는 *rel'* 릴레이션과 *R'* 릴레이션을 *a* 속성에 기반하여 조인함으로써 이루어진다.

[정리 2] 위에서 정의한 PISE 구현(그림 2)은 추출 연산이 기대하는 목표 스키마를 발생한다. 그리고 데이타 인스턴스 레벨에서도 추출 연산이 의도한 변화가 일어난다.

[증명] 릴레이션 *rel(k, a, b)*에서 새 릴레이션 *R(a, b)*가 추출되었고 릴레이션 *rel*은 속성 *b*가 빠진 릴레이션 *newRel*로 변환되었기 때문에 스키마 레벨에서는 릴레이션 *R*과 *newRel*로 이루어진 목표 스키마가 생성되었다. 피연산 릴레이션 *rel*의 인스턴스 *inst*는 속성 *k, a, b*에 대한 각각 v_k, v_a, v_b의 값을 가진다. 이 *inst* 인스턴스는 실제적으로 원천 릴레이션 *rel'*의 *inst'* 인스턴스에서 *k, a, b* 속성을 프로젝트하여 유도되었다. 실제 인스턴스 *inst'*은 속성 *k, a, b, c*에 대하여 각각 v_k, v_a, v_b, v_c의 값을 가지며 이는 위의 PISE 1단계에 의하여 두개의 튜플(v_k, v_a, v_c)와 (v_a, v_b)로 나누어지고 이들은 목표 스키마 유도함수에 의해 각각(v_k, v_a)와 (v_a, v_b)로 변환된다. 결과적으로 피 연산 인스턴스 *inst* = (v_k, v_a, v_b)는 (v_k, v_a)와 (v_a, v_b) 튜플들로 나누어지고 이는 추출 연산이 의도하는 데이타 인스턴스의 변화이다.

[정리 3] 위의 PISE 구현(그림 2)이 발생한 목표 릴레이션에 대한 생성, 갱신, 제거 연산은 목표 릴레이션이 베이스일 때와 같은 행동 (behavior)을 보여준다.

[증명] 릴레이션 *newRel*에 대한 인스턴스 생성은 *newRel'*에 대한 인스턴스 생성으로 변환되고 이는 역으로 *newRel* 릴레이션이 단지 *newRel'*에 대한 프로젝트 이므로 *newRel*에 새 인스턴스가 생성된 것으로 보여진다. 릴레이션 *R*에 대한 인스턴스 생성은 *R'*에 대한 인스턴스 생성으로 변환되고 마찬가지로 이는 *R*에 대한 인스턴스 생성으로 보여 진다. 릴레이션 *newRel*에서 튜플을 제거하면 그 튜플은 실제적으로 *newRel'*에서 제거

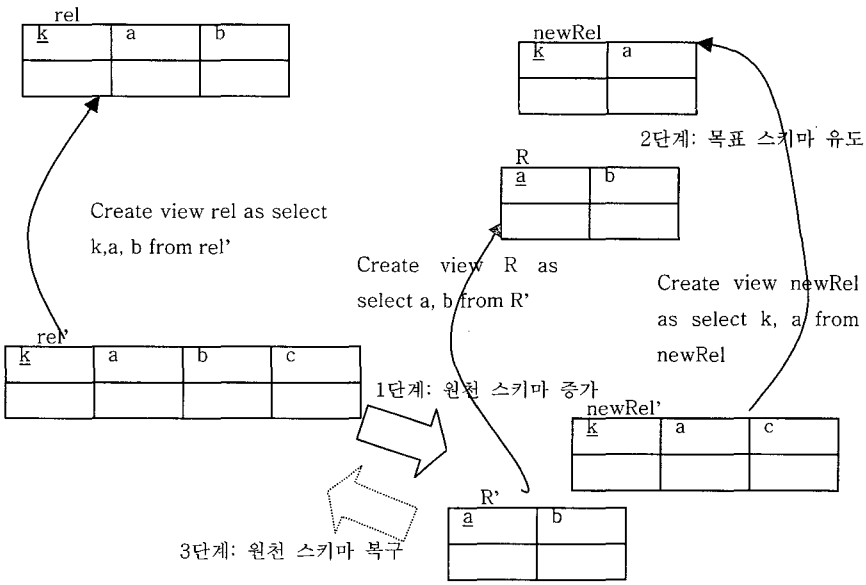


그림 2 추출 연산의 PISE 구현

되고 이는 *newRel*에서 그 튜플이 제거된 것처럼 보여진다. 릴레이션 *newRel*의 어떤 튜플을 갱신하면 이는 *newRel'*의 같은 튜플에 대한 갱신으로 변환되고 이것이 *newRel*에 전파되어 그 튜플이 *newRel*에서 갱신된 것으로 보여 진다. 이러한 생성, 제거, 갱신에 대한 행동(behavior)은 *newRel*이 베이스 릴레이션인 경우와 같다. 그러므로 *newRel*은 사용자에 베이스 릴레이션으로 보여 진다. 이는 *R* 릴레이션의 경우도 마찬가지다.

[정리 4] 위의 추출 연산의 PISE 구현(그림 2)은 원래 스키마 rel을 스키마와 인스턴스 레벨에서 복원하고 또한 이 복원된 스키마는 생성, 제거, 갱신 연산에 대해 베이스 스키마로 보여진다.

[증명] 원천 릴레이션 *rel'(k, a, b, c)*는 *newRel'(k, a, c)*와 *R'(a, b)*의 조인으로 복원되고 이에 대해 속성 *k, a, b*를 프로젝트하면 *rel(k, a, b)*를 스키마 레벨에서 복원된다. 릴레이션 *rel*의 튜플(v_k, v_a, v_b)은 실제적으로 *rel'*의 튜플(v_k, v_a, v_b, v_c)의 프로젝트이며 이 튜플은 물리적 스키마 변화(단계 1)에 의해 (v_k, v_a, v_c)와 (v_a, v_b)로 나누어지고 복원 단계(3 단계)에서 속성 a를 기반으로 조인되어 튜플(v_k, v_a, v_b, v_c)을 생성한다. 이 조인 튜플이 프로젝트되어 *rel*의 튜플(v_k, v_a, v_b)이 된다. 즉, 추출 연산에 의해 나누어진 *rel*의 튜플들은 3 단계에 의해 복원된다. 릴레이션 *rel*의 튜플(v_k, v_a, v_c) 생성은 *rel'*의 $(v_k, v_a, v_b, null)$ 인스턴스 생성으로 -숨겨진 속성 a에 대한 값은 null로 처리 - 변환되고 이는 다시 *newRel'*과 *R'*의 튜플$(v_k, v_a, null)$과 (v_a, v_b) 생성으로 변환된다. 즉, *newRel'*과 *R'*에 새 튜플들이 물리적으로 생성된다. 이 생성은 결과적으로 조인과 프로젝트 연산의 뷰 유도 체인을 거쳐 *rel* 릴레이션에 새 튜플(v_k, v_a, v_b)이 나타나게 한다. 즉, 릴레이션 *rel*에 새 튜플이 직접 생성된 것처럼 보이게 된다. 릴레이션 *rel*의 어떤 튜플을 제거하면 이 튜플에 해당하는 *newRel'*과 *R'*의 튜플들이 제거되고 이는 *rel*에서 그 튜플이 제거되는 결과를 낳는다. 릴레이션 *rel*의 어떤 튜플의 속성 값을 변화시키면 이 튜플에 해당하는 *newRel'*과 *R'*중의 한 튜플의 속성 값을 변화시켜 이 변화가 뷰 유도함수 체인을 따라 전파되면 *rel* 릴레이션의 튜플 값이 직접 변화된 것처럼 보여 지는 결과를 낳는다. 즉, 복원된 릴레이션 *rel*은 실제로는 뷰임에도 불구하고 생성, 제거, 갱신 연산에 대해 베이스 릴레이션과 같은 행동을 보여주기 때문에 *rel* 릴레이션을 사용하는 사용자는 스키마 변화 연산에도 불구하고 어떤 변화도 관찰하지 못한다.

[정리 5] 추출 연산의 PISE 구현(그림 2)은 프로그램 독립성의 원칙을 보장한다.

[증명] 그림 2에서 변화 이전의 스키마를 *VS1*, 이후의 스키마를 *VS2*라 한다. 먼저 *VS2* 튜플들에 대해 *VS1*에서 접근 가능한 범위를 조사한다. *VS2*의 *newRel* 릴레이션의 튜플들은 전부*VS1*의 *rel* 릴레이션의 튜플이 된다. 즉, *newRel*의 키 속성 *k*의 값들은 전부 *rel* 릴레이션에서 접근가능하다. 릴레이션 *R*의 튜플들은 *rel*의 튜플이 아니므로 모든 튜플이 *rel*에서 접근 가능할 필요가 없다. *newRel*과 *a* 속성 값을 공유하는 튜플만이 *rel*에서 접근 가능하다. 이들 즉 *a* 속성 값을 공유하는 *R*의 튜플들의 *b* 속성 값은 궁극적으로 *k* 속성 값에 의해

결정되어 *b* 속성 값의 불일치가 없기 때문에 *VS1*의 릴레이션 *rel*의 *b* 속성에서 접근 가능하다. 역으로 *VS1*의 튜플들을 *VS2*에서 접근 가능한 범위를 조사한다. *VS1*의 릴레이션 *rel*의 튜플들은 *VS2*의 *newRel*의 튜플들이 된다. 그러므로 *rel* 릴레이션 튜플들의 키 속성 *k* 값들은 *VS2*의 *newRel* 릴레이션에서 전부 접근 가능하다. 더욱이 릴레이션 *rel* 튜플들의 *a* 속성 값들은 전부 *VS2*의 *R* 릴레이션의 *a* 속성 값으로 접근 가능하다. 이는 *rel*의 *a* 속성에서 나타나는 값들은 조인 연산의 성질상 전부 *newRel*의 *a* 속성과 *R*의 *a* 속성의 공유된 값이야 하므로 성립한다. 릴레이션 *rel*의 *a* 속성 값은 *newRel*의 *a*의 속성을 통하여 접근 가능하다. *VS1*의 릴레이션 *rel*의 *b* 속성 값들은 *VS2*의 릴레이션 *R*의 키-속성 *a* 값에 대하여 유일하지 않을 수 있다. 이런 주키 제약조건에 불일치 되는 값들은 null로 지정된다. 이로써 *VS1*의 데이타 값들이 프로그램 독립성 원칙 하에서 *VS2*를 통하여 접근 가능하며 *VS2*의 데이타 값들 역시 마찬가지로 프로그램 독립성 원칙 하에서 *VS1*을 통하여 접근 가능함을 보였다.

*VS1*의 릴레이션 *rel*의 새 튜플의 생성은 릴레이션 *newRel'*의 튜플 생성으로 변환되고 이는 *newRel* 릴레이션의 튜플 생성으로 이어진다. 그러므로 *rel*의 *k* 키 속성 값이 새로 생기면 *newRel*에서 같은 *k* 키 속성 값이 생긴다. *VS2*의 릴레이션 *newRel*에서 새 *k* 키 속성 값이 발생하면 마찬가지 이유로 *VS1*의 릴레이션 *rel*에서 같은 *k* 키 속성 값이 생긴다. *VS2*의 릴레이션 *R*의 *a* 키 속성 값이 새로 생기면 이는 *newRel*의 어떤 튜플과 *a* 속성과 값을 공유하기 전에는 *VS1*의 *rel* 릴레이션의 *a* 속성 값을 새로 발생시키지 않는다. 이는 *rel* 릴레이션에서 *a* 속성이 키가 아니므로 프로그램 독립성의 원칙을 위반하지 않는다. *VS1*의 *rel* 릴레이션의 *a* 값의 생성은 원천 스키마의 *R'* 릴레이션의 *a* 값 생성으로 변환되고 이는 *R* 릴레이션의 *a* 값 생성으로 이어진다. 즉 *VS1*의 *rel* 릴레이션의 *a* 값 생성은 *VS2*의 *R* 릴레이션의 *a* 속성을 통하여 관찰된다. *VS1*의 *rel* 릴레이션 튜플에서 *k* 값을 제거하거나 갱신하면 이는 원천 베이스 스키마의 *newRel'* 릴레이션에서 *k* 값이 제거되거나 갱신된다. 이는 *VS2*의 *rel* 릴레이션 튜플이 제거되거나 *k* 값이 갱신되는 결과를 낳는다. 역으로 *VS2*에서 *newRel* 릴레이션의 *k* 값이 제거되거나 갱신되면 *VS1*의 *rel* 릴레이션에서 *k* 값이 제거되거나 갱신된다. *VS1*의 *rel* 릴레이션의 *a* 값이 제거되거나 갱신되면 원천 스키마의 *R'* 릴레이션의 *a* 값의 제거, 갱신으로 변환되고 이는 *VS2*의 *R* 릴레이션의 *a* 값의 제거, 갱신으로 전파된다. 즉, *VS1*의 키 속성 *k* 값을 *VS2*에서 변경하면 이는 *VS1*에서 보이고 역으로 *VS2*의 키 속성 *k*, *a* 값들을 *VS1*에서 변경하면 이 역시 *VS2*에서 관찰된다. 이는 키 값에 대한 프로그램 독립성의 원칙과 일치한다.

*VS2*의 릴레이션 *R*의 *a* 값이 제거, 갱신되면 이는 원천 스키마의 *R'*의 *a* 값의 제거, 갱신으로 이어진다. 그리고 이는 *VS1*로 전파되어 *rel*의 *b* 값에 *null*이 할당될 수도 있다. 이는 *VS2*에서 *b* 속성 값이 키 속성 *k*에 의존하지 않게 되어 생기는 현상으로 프로그램 독립성을 해치지 않는다. *VS1*에서 *rel* 릴레이션의 *b* 속성 값을 변화시키면 이는 *VS2*의 *R* 릴레이션의 *b* 값 변화로 전파된다. 이때 *b* 값 변화가 b 값의 a 값에 대한 의존성을 위반하면 *R*의 *b* 값은 *null*로 지정된다. 즉, 이 경우- *b* 값이 주키 제약 조건을 해치는 - *VS1*에서의 *b* 값 변화는 *VS2*에 전파되지 못한다. *VS2*에서 *R* 릴레이션의 *b* 값을 갱신하는 경우 이 튜플의 *a* 값이 *newRel*의 어떤 튜플과 *a* 값을 공유하면 이 변화는 *VS1*의 *rel* 릴레이션의 *b* 속성으로 전파된다. 그러나 변화된 튜플의 *a* 값이 *newRel*의 어떤 튜플과도 *a* 값을 공유하지 못하면 *VS2*의 *b* 값 변화는 *VS1*의 *rel* 릴에이션에서 보이지 않는다. 이 역시 *VS1*의 릴레이션 *rel*의 *b* 속성에 해당하는 다른 스키마 버전의 속성 값이 *k* 키 속성 값에 대한 의존성을 위반하여 생기는 현상으로 프로그램 독립성을 해치지 않는다.

3.2 종속성-도입(import-dependency)

이 연산의 문법은 "*종속성-도입(rel, R)*" 이다. 여기서 *rel*과 *R*은 릴레이션들이다. 의미는 릴레이션 *rel*에 *R*의 비-키 속성들을 더하는 것이다. 전제 조건으로 *R*과 *rel*의 속성들이 같아야 한다. 인스턴스 레벨에서는 릴레이션 *R*의 인스턴스들의 비-키 속성 값들이 그 인스턴스들과 R의 키 값들을 공유하는 릴레이션 *rel*의 인스턴스들에 덧붙여진다.

이 연산은 비-용량-증가이다. 왜냐하면 이 연산의 역인 추출이 용량-증가이기 때문이다. PISE 방법론에 따르면 이 연산이 비-용량-증가이기 때문에 목표 스키마는 피연산 릴레이션으로부터의 뷰로써 생성된다. 그림 3에서 목표 뷰, *newRel* 릴레이션, 은 그림 3의 예에서 보듯이 *R*과 *rel*의 키 속성 a를 기반으로 릴레이션 *rel*과 *R*을 조인함으로써 생성된다. 인스턴스 레벨에서는 *a* 키 속성 값들을 공유하는 *rel*과 *R* 릴레이션의 모든 인스턴스들은 결합되어 목표 릴레이션 *newRel*의 인스턴스들이 된다.

3.3 이름 변화(change-name)

이 연산은 릴레이션 혹은 속성의 이름을 변화한다. 이 연산의 문법은 "*이름-변화(oldName, newName)*"이다. 이 연산의 의미는 릴레이션 혹은 속성의 이름을 *oldName*으로부터 *newName*으로 바꾸는 것이다.

이 연산은 스키마의 구조를 변화시키지 않고 새로운

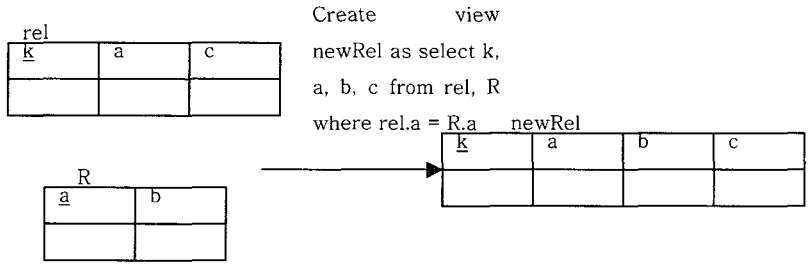


그림 3 종속성-도입 연산의 뷰 접근 방식 구현

스키마 요소를 추가하지 않으므로 비-용량-증가이다. PISE 방법론에 의하면(전의 개괄 절에서 보여 지었듯이) 이름-변화 연산은 비-용량-증가이기 때문에 목표 뷰를 생성함으로써 달성된다. 목표 뷰 생성은 그림 4의 오른쪽에 보여진다. 그림 4에서 보듯이 *옛릴레이션*으로 이름 붙여진 릴레이션은 뷰 정의 함수를 이용하여 *새릴레이션*으로 이름을 바꾼다. *옛속성*으로 이름 붙여진 속성은 그림 4의 뷰 정의에서 이름이 *새속성*으로 바뀌어진다.

3.4 속성 더하기(add-attribute)

이 연산의 문법은 "*속성-더하기(newAttr, rel)*"이다. 의미는 릴레이션 *rel*에 속성 *newAttr*을 더하는 것이다. 데이타 인스턴스 레벨에서 릴레이션 *rel*은 모든 값이 null로 지정되는 새 속성 *newAttr*을 갖는다.

이 연산은 새로운 스키마 요소 즉, 속성을 추가하므로 용량-증가이다. PISE 방법론에 의하면 이 연산은 첫째로 기존의 릴레이션 *rel*을 바꿔 그림 5(b)에서 보듯이 새 속성 newAttr을 갖게 한다. 릴레이션 *rel*이 비록 베이스 릴레이션으로 보여 지지만 실제로는 *rel'* 릴레이션으로부터의 뷰로써 구현됐음을 주의하라. 목표 릴레이션 *newRel*은 그림 5(c)에서 보여지듯이 원천 릴레이션 *newRel'* 으로부터의 뷰로써 생성된다.

3.5 속성 지우기(delete-attribute)

이 연산의 문법은 "*속성-지우기(attr, rel)*"이다. 의미는 릴레이션 *rel*으로부터 속성 *attr*을 지우는 것이다. 데

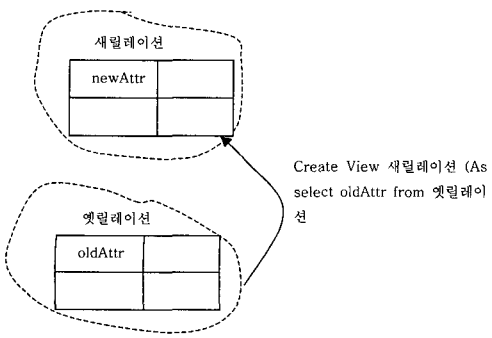


그림 4 이름-변화 연산의 뷰 생성

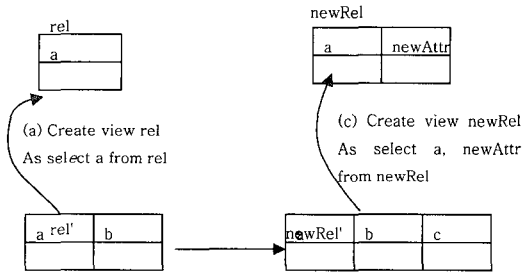


그림 5 속성 더하기 연산의 PISE 구현

이타 인스턴스 레벨에서는 *attr* 값에 해당하는 속성의 저장 값이 버려진다.

이 연산은 기존의 스키마 요소, 즉 속성을 제거하기 때문에 비-용량-증가이다. PISE 방법론에 따르면 이 속성-지우기 연산은 연산의 의미를 반영하는 목표 뷰를 생성한다. 속성 지우기 경우에서 그림 6에서 보듯이 *newRel* 릴레이션은 피연산 릴레이션 *rel*으로부터의 뷰로써 생성된다. 목표 릴레이션 newRel은 원천 릴레이션 rel으로부터의 뷰로써 정의된다.

그림 6 속성 지우기 연산

3.6 키 상승(promote-key)

이 연산의 문법은 "*키-상승(attr, rel)*"이다. 의미는 *rel* 릴레이션의 키를 재정의하여 *attr* 속성이 *rel* 릴레이션의 키에 포함되도록 하는 것이다. *Rel* 릴레이션의 인스턴스들은 그들의 *attr* 값이 null로 지정된다. 새 키 값의 유일성 제약 조건은 새 속성이 키에 더해지는 것이므로 위반될 수 없다.

이 연산은 *rel* 릴레이션의 주키 제약이 키에 속성 *attr*이 추가적으로 더해져, 약해지기 때문에 용량-증가이다. 그림 7(a)에서 보듯이 피연산자 관계 *rel*은 실제

로는 주 키 제약조건 *pk(a)*에 제약을 받는 뷰 릴레이션이다. 원천 베이스 릴레이션 *rel'*은 직접적으로 변화하여 속성 *attr*이 그림 7(b)의 *newRel'*의 주키가 된다. 목표 릴레이션 *newRel*은 그림 7(c)에서 보여 지듯이 주키 제약조건이 속성 *attr*을 포함하는 뷰로써 생성된다.

3.7 키 강등(demote-key)

이 연산의 문법은 "*키-강등(attr, rel)*"이다. 의미는 *attr* 속성이 *rel* 릴레이션의 키로부터 제외되는 것이다. 이 연산은 결과적으로 줄어든 새 키가 튜플을 유일하게 결정하는데 충분한지 아닌지를 확인하기 위하여 *rel* 릴레이션의 튜플들을 조사하도록 요구한다. 두개 이상의 튜플이 같은 목표 키 값을 공유한다면 DBA는 목표로 하는 *newRel* 저장 데이타베이스가 유일성 제약조건을 위반하지 않도록 스키마 진화 시스템과 상호작용을 하여야 한다.

이 연산은 비-용량-증가이다. 왜냐하면 *attr* 속성을 제외하여 주키 제약 조건이 강화되기 때문이다.

이 연산의 구현은 그림 8에 보여진다. 속성 *attr* 이 그림 8의 오른쪽에서 보여 지듯이 비-키가 되도록 뷰 릴레이션 *rel'* 이 발생된다. 그림 8의 오른쪽의 튜플들의 *a* 값이 유일해야 한다. 반면에 그림 8의 왼쪽 릴레이션 튜플들의 *a* 값은 *attr* 값이 다른 한에서는 유일할 필요가 없다. 목표 뷰는 원천 릴레이션의 직접적인 변화 없이 피연산자 릴레이션으로부터 직접적으로 생성된다. 그림 8에서는 속성 *attr*은 강등되어 *rel'* 릴레이션이 속성 *a* 로만 이루어진 주키를 갖게 된다. 이는 *rel'* 릴레이션의 주키 제약조건에서 *attr*을 삭제함으로써 실현된다.

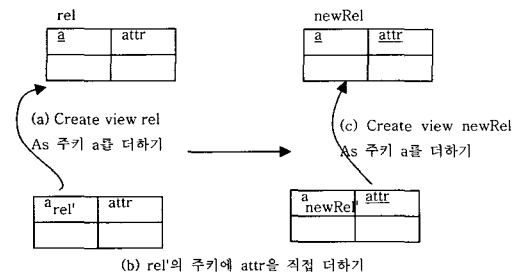


그림 7 키 상승 연산

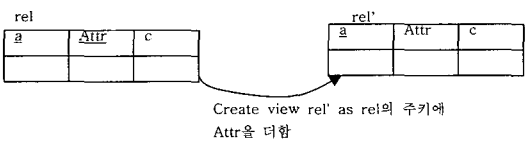


그림 8 키 강등 연산

3.8 릴레이션 분해(decompose)

이 연산의 문법은 "*분해(rel, R1(k1, k2, ..., kl, a1, a2, ..., am), R2(k1, k2, ..., kl, b1, b2, ..., bn))*"이다. 의미는 릴레이션 *rel*을 각각 속성이 $k_1, k_2, ..., k_l, a_1, a_2, ..., a_m$과 $k_1, k_2, ..., k_l, b_1, b_2, ..., b_n$인 두개의 릴레이션 *R1*과 *R2*로 매핑되도록 한다. 두개의 릴레이션 *R1*과 *R2*는 $k_1, k_2, ..., k_l$인 같은 키를 공유한다. 데이타 인스턴스 레벨에서는 *rel*의 인스턴스들이 두 개의 인스턴스들 즉, *R1*과 *R2*의 인스턴스들로 나누어진다.

이 연산은 용량-증가이다. R1 (R2)에 속하는 인스턴스들과 주키 값을 공유하는 인스턴스들이 반대편 릴레이션 R2 (R1)에 없는 경우가 있다. 이때 그러한 인스턴스들은 *rel* 릴레이션에 속할 수 없고 그래서 *rel* 릴레이션의 용량은 *R1*과 *R2*를 합한 것 보다는 적다.

초기에 *rel* 릴레이션은 그림 9의 왼쪽에 보여지듯이 베이스 릴레이션 *rel'*으로부터의 뷰이다. 이 연산은 용량-증가이기 때문에 PISE 방법론에 의하면 원천 베이스 릴레이션 *rel'*은 그림 9의 아래 부분에 보여 지듯이 물리적으로 *R1'*과 *R2'*으로 나누어진다. 다음에 그림 9의 오른쪽에서 보여 지듯이 목표 릴레이션 *R1*과 *R2*는 각각 베이스 릴레이션 *R1'*과 *R2'*으로부터의 뷰로써 발생된다.

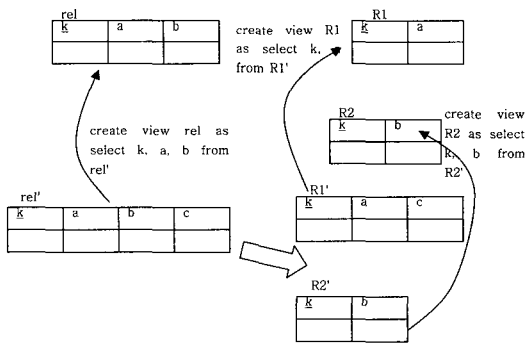


그림 9 분해 연산의 PISE 구현

3.9 릴레이션 결합(compose)

이 연산의 문법은 "*결합(R1, R2, rel)*"이다. 의미는 릴레이션 *R1*과 *R2*를 릴레이션 rel로 매핑하는 것이다. 전제 조건으로 *R1*과 *R2* 릴레이션의 키 속성들은 같아야 한다. 릴레이션 *rel*의 주키는 *R1* 혹은 *R2*의 주키와 같다. 릴레이션 *rel*의 비-주키 속성은 *R1*과 *R2*의 비-주키 속성들의 합이다. 같은 키 속성 값을 갖는 *R1*과 *R2* 인스턴스들은 rel 인스턴스들로 결합된다.

이 결합 연산은 비-용량-증가이다. 왜냐하면 이 연산의 역 (inverse)인 분해 연산이 용량-증가이기 때문이다. 결합 연산은 그림 10에서 보여 지듯이 뷰 접근 방식으로 달성된다. 목표 릴레이션 *rel*은 주키 속성을 기반으로 피연산자 릴레이션 *R1*과 *R2*를 조인하는 뷰로써

생성된다. 인스턴스 레벨에서 같은 키 속성 k 값을 갖는 *R1*과 *R2*의 인스턴스들이 조인되어 릴레이션 *rel*의 인스턴스들이 된다.

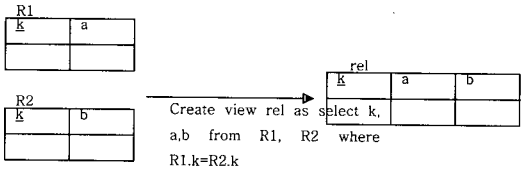


그림 10 결합 연산의 뷰 접근 방식

3.10 릴레이션 나누기(partition)

이 연산의 문법은 "*나누기(rel, c1, R1, c2, R2)*"이다. 여기서 *c1*과 *c2*는 *rel*의 인스턴스들이 릴레이션 각각 *R1*과 *R2*의 인스턴스들로 나누어질 때 사용되는 조건이다. 의미는 릴레이션 *rel*의 튜플들을 제약 조건 *c1*과 *c2*에 따라서 릴레이션 *R1*과 *R2*에 나누어 넣는 것이다. 목표 릴레이션 *R1*과 *R2*는 나누어지는 릴레이션 *rel*과 같은 속성을 가져야 한다. 제약 조건 *c1*과 *c2*는 원천 릴레이션 *rel*의 모든 튜플에 대하여 참 또는 거짓으로 평가되어야 하는 불리언 (boolean) 조건들이다. 릴레이션 *rel*의 각 튜플 *r*은 제약조건 *c1*과 *c2*를 평가하여 *R1* 또는 *R2* 릴레이션의 튜플들로 나누어진다. 만약 *r*에 대한 c1/c2가 참이라면 r은 *R1/R2* 각각으로 분류된다. 목표 스키마는 릴레이션 *rel*의 정의를 릴레이션 *R1*과 *R2*의 합집합으로 정의함으로써 얻어진다.

이 연산은 비-용량-증가이다. 그래서 PISE 방법론에 따르면 목표 스키마는 뷰 유도함를 사용하여 목표 릴레이션을 생성함으로써 얻어진다. 그림 11에서 목표 릴레이션 *R1(R2)*는 뷰 정의 "select * from rel where c1 (c2)"에 의해 생성된다. 릴레이션 *rel*의 인스턴스들은 조건 *c1* 혹은 *c2*를 만족시키느냐에 따라 R1 또는 R2로 이동한다.

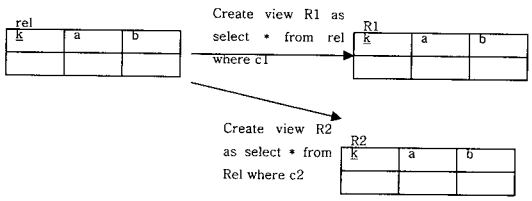


그림 11 나누기 연산을 위한 뷰 접근 방식

3.11 릴레이션 합병(merge)

이 연산의 문법은 "*합병(rel, R1, R2)*" 이다. 의미는 R1과 R2 릴레이션의 튜플을 결합하여 rel 릴레이션에 넣는 것이다. 전제조건으로 R1, R2, 그리고 rel 릴레이션은 같은 속성을 갖는다. 합병이 수행되기 전에 튜플들 목표 릴레이션에서의 유일성을 보장하기 위하여 튜플들이 R1, R2, 그리고 rel 릴레이션 중 어느 하나 이상에서 나타나지 않도록 확인을 해야 한다.

이 연산은 *R1* 혹은 *R2*에 분류되지 않는 *rel*의 인스턴스들이 있을 수 없기 때문에 비-용량-증가이다. 그래서 목표 릴레이션 rel은 그림 12에서 보듯이 릴레이션 R1과 R2의 합집합으로 정의되는 뷰로써 얻어진다. 이 합병 연산에서 R1과 R2의 속성들은 서로 같고 *rel* 릴레이션의 속성들과도 같아야 한다. *R1*과 *R2* 릴레이션의 모든 인스턴스들은 *rel* 릴레이션에서도 나타난다. 이러한 내포적인 그리고 외연적인 의미는 그림 12에서 보듯이 목표 뷰 *rel*을 피연산 릴레이션 *R1*과 *R2*의 합집합으로 정의함으로써 달성된다.

3.12 부분 의존성을 도출(export)

이 연산의 문법은 "*도출(rel, R(k_1, k_2, ..., k_n, a_1, a_2, ..., a_n))*"이다. 여기서 k_1, k_2, ..., k_n 속성들은 *rel* 릴레이션의 키 속성들의 부분집합이고 a_1, a_2, ..., a_n은 rel 릴레이션의 비-키 속성들의 부분집합이다. 의미는 릴레이션 rel의 키 속성의 부분집합 k_1, k_2, ..., k_n과 비-키 속성의 부분집합 a_1, a_2, ..., a_n을 도출하여 새 릴레이션 R을 구성하는 것이다. 릴레이션 rel에서 속성들 a_1, a_2, ..., a_n이 제거된다. 이 릴레이션은 *newRel*이라하여 *newRel*은 부분 의존성이 없이 4번째 정규형(normal) 폼이 된다.

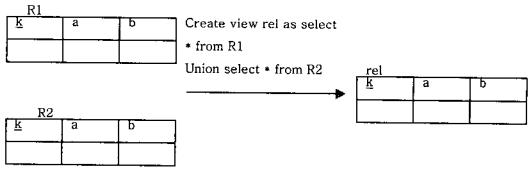


그림 12 합병 연산을 위한 뷰 접근 방식

릴레이션 *R*의 인스턴스들과 키 속성 값을 공유하지 않은 *newRel*에 속하는 인스턴스들이 있을 수 있다. 게다가 *newRel* 인스턴스들과 키 속성 값을 공유하지 않은 *R*에 속하는 인스턴스들이 있을 수 있다. 그러한 *R*과 *newRel*의 인스턴스들은 피연산 릴레이션 rel에 나타날 수 없고 그래서 이 연산은 용량-증가이다.

PISE 방법론에 따르면, 피연산 릴레이션 *rel*은 그림 13의 왼쪽에서 보듯이 원천 베이스 릴레이션 *rel'*으로부터의 뷰로써 정의된다. 원천 릴레이션 *rel'*은 그림 13의 아래에 블록 화살표로 나타났듯이 두개의 릴레이션 *R'*과 *newRel'*으로 물리적으로 나누어진다. 목표 릴레이션 *newRel*과 R은 원천 릴레이션 *newRel'*과 *R'*으로부터 뷰로써 각각 생성된다. 초기 릴레이션 *rel*은 릴레이션 *rel'*에 속한 어떤 속성들을 프로젝트하는 뷰로써 정의된

다. 릴레이션 *newRel* 역시 릴레이션 *newRel'*에 속하는 속성들을 프로젝트하는 뷰이다.

3.13 도입(import)

이 연산의 문법은 *"도입(rel1, rel2)"* 이다. 전제 조건으로 릴레이션 *rel2*의 키 속성들은 릴레이션 *rel1* 키 속성들의 부분 집합이다. *Rel2*의 비-키 속성들은 릴레이션 *rel1*에 더해진다. PISE 방법론에 따르면 이 연산은 피 연산 릴레이션을 조인하는 목표 뷰를 생성함으로써 구현된다. 그림 14는 결과 스키마를 적절한 뷰로써 생성함으로 이 도입 연산의 의미를 어떻게 달성하는가를 보여준다. 목표 릴레이션 *rel1'*은 키 속성 *rel1.k*와 *rel2.k*를 기반으로 릴레이션 *rel1*과 *rel2*를 조인하는 뷰로써 생성된다.

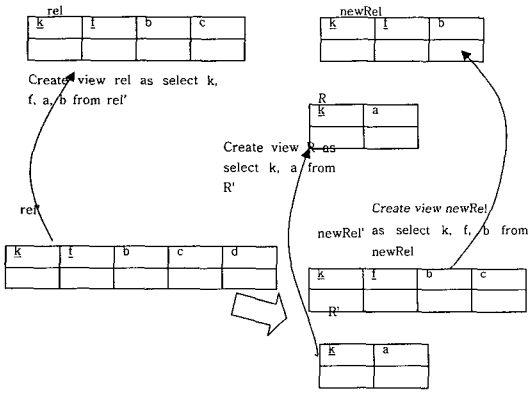


그림 13 도출 연산의 PISE 구현

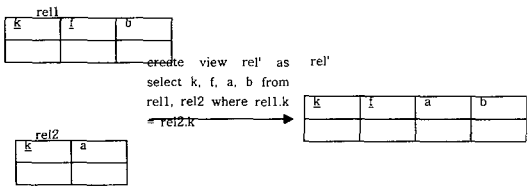


그림 14 도입 연산의 뷰 접근 방식

4. 관련 연구

스키마 변화 접근 방식은 네 개의 카테고리로 분류된다. 첫째 카테고리는 통상의 스키마 변화 개념을 지원하는 *직접적인* 스키마 변화 시스템에 해당한다. 즉, 공유된 베이스 스키마를 직접적으로 변화하고 이를 인스턴스 레벨에 전파한다. 두 번째 카테고리는 통상적인 버전 개념을 지원하는 스키마 버전 시스템에 해당한다, 즉, 옛 스키마가 단지 질의 목적을 위해서 데이타베이스의 스냅 샷과 함께 저장된다. 세 번째 카테고리는 프로시저 프로그램 독립적인 변화 시스템으로 프로그램 독립성 개념을 지원한다. 즉, 옛 스키마의 응용 프로그램은 스키마 변화에도 불구하고 지속적으로 옛 스키마에 동작하고 인스턴스와 스키마 버전과의 타입 불일치는 프로그래밍 언어로 쓰인 프로시저에 의해 해소된다. 네 번째 카테고리는 뷰-기반의 스키마 변화 시스템으로 역시 프로그램 독립성 개념을 지원하나 차이는 각 스키마 버전을 베이스 스키마의 뷰로써 구현함으로써 타입 불일치를 해소한다. 반면에 세 번째 시스템에서는 불일치가 사용자/미리-정의된 프로시저에 의해 해소된다.

이 논문의 관심이 스키마 변화의 프로그램 독립성에 있기 때문에 세 번째와 네 번째 시스템에 집중한다. 다음의 절은 프로시저 프로그램 독립적인 시스템과 뷰-기반의 프로그램 독립적인 시스템을 소개하고 PISE 시스템과 비교하여 장단점을 논한다.

4.1 프로시저 프로그램 독립적인 시스템

이 카테고리의 시스템은 스키마의 변화의 프로그램 독립성 개념을 지원한다, 즉, 옛 응용 프로그램이 스키마 변화에도 불구하고 지속적으로 지원된다. 이 프로그램 독립성은 모든 스키마 버전을 위해 단지 하나의 데이타 복사본만 유지함으로써 달성된다. 저장된 인스턴스와 스키마의 특정 버전과의 타입 불일치는 인스턴스들이 접근될 때 적절한 프로시저를 적용함으로써 해소된다.

4.1.1 인코어(encore) 시스템

Zdonik과 그 외의 인코어(encore) 시스템[12]에서의 타입 변화에 대한 접근방식은 각 타입을 위해 인스턴스들이 생성될 때 인스턴스들을 타입의 특정 버전에 묶는다(bind). 다른 타입 버전의 인스턴스들은 그 인스턴스들의 타입이 포함하지 않는 성질을 위한 예외 처리기(exception handler)를 제공함으로써 접근될 수 있다. 새로운 속성이 타입에 더해질 때 타입의 모든 옛 버전에 새 스키마로부터 옛 인스턴스들의 새 속성 값들이 (초기에 정의되어 있지 않은) 접근되는 경우를 위한 예외 처리기가 제공되어야 한다. 관리자가 의미 있는 예외 처리기를 제공하는 것은 노동이 많이 들뿐만 아니라 어려운 일이다. 특히 오래 전에 다른 개발자에 의해 설정된 타입을 위해서는 더욱 그렇다.

4.1.2 CLOSQL

Monk[14]에 의한 CLOSQL은 각 속성을 위해 갱신/소급 함수가 제공되어(사용자 혹은 어떤 연산을 위해 미리 정의된) 인스턴스들이 저장된 형식으로부터 응용 프로그램이 기대하는 형식으로 변환하게 할 것을 제안한다. 그러한 시스템에서 디폴트(default) 변환 함수가 제공되지 않는다면 사용자의 책임은 더욱 커질 것이다.

4.1.3 Clamen의 접근방식

Clamen[20]은 프로그램 독립성 이슈에 대안을 제시한다. 그의 연구는 위에서 설명한 Zdonik의 타입 변화 관

리[12]의 일반화이다. Clamen의 안에 의하면 인스턴스의 물리적 표현은 모든 버전의 중첩되지 않은 합집합이다. 반면에 Zdonik의 안에서는 인스턴스의 물리적 표현은 모든 버전의 최소 커버(cover)인 인터페이스 타입(interface type)으로 이루어진다. 인스턴스 물리적 표현의 이러한 일반화는 인스턴스와 버전간의 더욱 유연한 인스턴스 적응을 가능하게 한다. 그러나 인스턴스의 표현은 새로운 버전이 생성될 때마다 상당히 증가되어야 한다. 이는 저장과 동작에 있어서 받아들일 수 없는 부담으로 이어질 수 있다.

4.2 뷰-기반의 프로그램독립적인 스키마 변화 시스템

이 카테고리의 시스템은 이 시스템들이 여러 스키마 버전을 위해 단지 하나의 저장 데이타베이스만 유지한다는 데서 프로시저 프로그램 독립적인 스키마 변화 시스템과 같다. 그러나 이 시스템은 특정한 스키마 버전에 상응하는 데이타베이스의 시각을 유도함으로써 타입 불일치를 해소한다. 이 시각은 데이타베이스 뷰로 구현된다. 그래서 잘 정의된 언어(SQL)가 유도에 사용될 수 있다. 그리고 어떻게 효율적으로 변경된 데이타베이스 뷰의 외연을유도할 것인가 그리고 어떻게 갱신을 할 것인가에 대한 많은 기존의 지식을 적용할 수 있다.

4.2.1 Tresch와 Scholl의 접근방식

Tresch and Scholl[16]은 스키마 진화를 달성할 수 있는 적절한 메커니즘으로 뷰를 옹호한다. 그들은 진화가 용량-증가가 아니면 스키마 진화는 뷰를 이용해서 달성될 수 있다고 한다. PISE 방법론은 직접 변화와 뷰 시뮬레이션 접근방식을 통합함으로써 이 한계를 극복한다.

4.2.2 Bertino의 접근방식

Bertino[15]는 역시 뷰 메커니즘을 소개하여 그것이 스키마 진화를 달성하는 데 활용될 수 있다고 언급한다. 제안된 메커니즘은 새 저장 속성이 뷰에 더해질 수 있다는 데서 용량-증가이다. 그러나 유연한 구조조정을 어떻게 대처하느냐 하는 구현 이슈는 논의되지 않았다.

4.2.3 Thomas와 Shneiderman의 접근방식

Thomas와 Shneiderman은 네트워크 데이타 모델에서 프로그램 독립적인 스키마 변화를 위한 뷰-기반 해결책을 제안한다[21]. 이들은 PISE 방법론이 뷰로써 옛 스키마를 유지하는데 반해 옛 스키마를 네트워크 서브-스키마로 구현할 것을 제안한다. 그래서 기저가 되는 데이타 모델이 다르다는 것을 제외하고는 옛 스키마를 유도하는 아이디어는 두 접근방식 모두 비슷하다.

그러나 이 시스템은 직접 스키마변화 연산자를 지원하고 옛 스키마는 뷰가 아니라 베이스 스키마이기 때문에 옛 스키마는 주요 요구사항 변화에도 더 이상 변화되지 않는다. 그래서 옛 스키마는 더 이상의 변화에 대해 고정된다.

5. 결론과 미래 연구

이 논문은기존의 프로그램에 영향을 미치는 스키마 진화 문제에 해결책을 소개한다. 어떤 주어진 스키마에 대한 변화가 다른 스키마들에, 비록 그들이 공통된 베이스 스키마를 가지고 있고 데이타 저장소를 공유한다 할지라도, 영향을 미쳐서는 안 된다. 더욱이 기존의 프로그램을 계속 지원하기 위해서는 옛 스키마가 보존되어야 하고 데이타를 최신 상태로 유지되어야 한다. 위의 조건을 만족시키는 스키마 변화연산을 프로그램 독립적이라고 한다.

프로그램 독립적인 스키마 변화 연산은 각 개발자가 다른 사람과 같은 데이타베이스를 공유하면서 필요에 따라 그의 스키마를 변화시킬 수 있도록 보장한다. 더욱이 중요한 것은 레거시 소프트웨어 시스템이 스키마가 변경되는 동안에도 동작을 지속할 수 있게 하는 부드러운 이동 경로를 제공한다.

PISE 접근방식의 유용성과 실용성을 보여주기 위해 이 논문은 포괄적인 스키마 변화 연산 집합을 선택하였다. 이 집합은 문헌[1]에서 발견한 스키마 변화 연산의 대부분을 포함한다. 이 논문 작업의 한 중요 결과는 PISE 메커니즘의 틀 안에서 이 집합의 연산들 모두가 "프로그램 독립적으로 달성될 수 있음"을 보이는 데 있다.

서론 절에서 언급하였듯이 관계형 뷰를 사용한 PISE 스키마 진화 접근방식은 스키마 진화와 뷰의 완전한 통합을 위한 경로를 보여주었다. 두 기능은 그들이 사용자에게 그들 고유의 목표를 위해 스키마 구조조정 역량을 제공하였다는 데서 똑같다. 그러나 뷰의 주요 역할이 주어진 스키마로부터 유도된 맞춤 인터페이스를 유도하는 것인데 반해 스키마 진화의 주요 역할은 스키마를 확장하는데 있다는 데서 두 메커니즘의 차이가 있다. 통합 목표를 위한 첫째 단계는 스키마 진화 연산이 임의의 사용자-정의 뷰 스키마에 적용되는 것을 허락하는 것이다. 그러면 스키마 진화와 뷰 기능 모두를 바꿔가면서 실행시킬 수 있게 된다.

PISE 시스템의 사용자 스키마는 실제로 뷰 스키마이기 때문에 PISE 시스템의 실행 능력은 지원하는 뷰 시스템의 실행 능력과 밀접하게 연관되어 있다. 그러나 최악의 경우에 PISE가 오랫동안 동작하여 지나치게 긴 유도 체인이 생길 수도 있다. 예를 들어 같은 릴레이션에 속성-지우기 연산을 열 번 연속으로 수행한다면 (실제 상황에서는 있을 것 같지는 않지만) 열 개의 유도가 있는 체인을 갖게 된다. 릴레이션의 가장 최신 버전에 질의와 변경은 베이스 릴레이션을 만나기 전까지 체인의 중간 릴레이션에 대한 여러 번의 질의/변경을 요구할 것이다. 이는 질의/변경의 실행 능력 저하를 가져온

다. 그래서 유도 체인을 제거하는 체계적인 방법이 연구되고 있다. 뷰가 저장 질의로 정의된다는 즉, 뷰에 대한 질의의 최적화는 뷰 정의가 서브-질의로 끼워 넣어지는 네스티드 (nested) 질의의 최적화와 동일하다는 것을 고려하면 이 문제 역시 질의 최적화를 사용해서 해결할 수 있음을 알 수 있다. 통상의 질의 최적화 기술이 어떻게 긴 유도 체인에 기인한 실행 능력 저하 문제를 대처하는데 사용될 수 있는 가를 알기 위해서는 더 이상의 조사가 필요하다.

참 고 문 헌

[1] Ben Shneiderman and Glenn Thomas, "An architecture for automatic relational database system conversion," *ACM Transactions on Database Systems* Vol. 7, No. 2, pp. 235-257, 1982.

[2] Susanne Busse and Claudia Pons, "Schema evolution in Federated Information Systems," *Database Systems in Business, Technology and Web*, pp. 26-43, 2001.

[3] Peter McBreien and Alexandra Poulovassilis, "Schema evolution in heterogeneous database architecture, a schema transformation approach," *Conference on Advanced Information Systems Engineering*, pp. 484-499, 2002.

[4] J. H. Jahnke, U. A. Nickel, and D. Wagenblasst, "A case study in supporting Schema Evolution of Complex Engineering Information Systems," *International Computer Software and Applications Conference*, pp. 513-521, 1988.

[5] Axel Wienberg, Matthias Ernst, Andreas Gawecki, Olaf Kummer, Frank Wienberg, and Joachim W. Schmidt, "Content schema evolution in the CoreMedia," *International Conference on Extending Database Technology*, pp. 712-721, 2002.

[6] Lex Wedemeijer, "Defining Metrics for Conceptual Schema Evolution," *Workshops on Foundations of Models and Languages for Data and Objects*, pp. 2210-244, 2000.

[7] Can Turker, "Schema evolution in SQL-99 and Commercial (Object-) Relational DBMS," *Workshops on Foundations of Models and Languages for Data and Objects*, pp. 1-32, 2000.

[8] Adriana Marotta, Regina Motz, and Paul Ruggia, "Managing source schema evolution," *Web Warehouses. Workshop on Information Integration on the Web*, pp. 148-155, 2001.

[9] Henderik Alex Proper, "Data schema design as a schema evolution process," *Data and Knowledge Engineering, Vol.* 22, No. 2, pp. 159-189, 1997.

[10] Zohra Bellahsene, "Schema evolution in data warehouses," *Knowledge and Information Systems* Vol. 4, No. 3, pp. 283-304, 2002.

[11] Edelweiss and Clesio Saraiva dos Santos, "Dynamic schema evolution management using version in temporal object-oriented databases," *International Workshop on Database and Expert Systems Application*, pp. 524-533, 2002.

[12] A. H. Skarra and S. B. Zdonik, "The management of changing types in object-oriented databases," *Proc. 1st Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 483-494, 1986.

[13] A. Mehta, D. L. Spooner and M. Hardwick, "Resolution of type mismatches in an engineering persistent object system," *Tech. Report, Computer Science Dept., Rensselaer Polytechnic Institute*, 1993.

[14] S. Monk and I. Sommerville, "Schema evolution in oodbs using class versioning," *SIGMOD RECORD, Vol.* 22, No. 3, 1993.

[15] E. Bertino, "A view mechanism for object-oriented databases," *3rd International Conference on Extending Database Technology*, pp. 136-151, 1992.

[16] M. Tresch and M. H. Scholl, "Schema transformation without database reorganization," *SIGMOD RECORD*, pp. 21-27, 1992.

[17] J. Ullman, *"Principle of Database Systems and Knowledge-Based Systems,"* Vol. 1, Computer Science Press, 1988.

[18] Y. G. Ra and E. A. Rundensteiner, "A transparent schema evolution system based on object-oriented view technology," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 4, 1997.

[19] W. Kim and H. Chou, "Versions of schema for OODB", *Proc. 14th Very large Databases Conference*, pp. 149-159, 1988.

[20] S. M. Clamen, "Type evolution and instance adaptation," *Technical Report CMU-CS-92-133R, Carnegie Mellon University, School of Computer Science*, 1992.

[21] G. Thomas and B. Shneiderman, "Automatic database system conversion: A transformation language approach to sub-schema implementation," *IEEE Computer Software and Applications Conference*, pp. 80-88, 1980.

나 영 국

1987년 서울대학교 전자공학과 졸업(학사). 1989년 The Penn. State Univ. 컴퓨터공학과 졸업(석사). 1996년 The Univ. of Michigan(Ann Atrbor) Comp. Sci. 졸업(박사). 1997년~1999년 삼성 SDS 재직. 1999년~2002년 국립한경대학교 재직. 2003년~현재 서울시립대학교 재직. 관심분야는 데이타베이스, XML, 스키마 진화 등