데이터베이스: 강의노트 04

A. Silberschatz, H. Korth, S. Sudarshan
Database System Concepts,
Fourth Edition, McGraw-Hill, 2002.

Part II. Relational Databases

4 SQL

4.1 배경

- IBM은 1970년대 초반에 Sequel이라는 질의어를 개발하였다.
- 그 이후 계속 발전되다가 SQL(Structured Query Language)로 이름이 바뀌었다.
- 1986년 ANSI와 ISO는 SQL-86이라고 하는 SQL 표준을 발표하였다.
- 이 표준은 계속 확장되어 SQL-89, SQL-92를 거쳐 현재 버전은 SQL:1999이다.
- SQL의 구성요소
  - 데이터 정의 언어: 관계 스키마 정의, 관계 삭제, 관계 스키마 수정
  - 대화식 데이터 조작 언어: 관계 대수 기반한 질의어와 투플 관계 해석에 기반한 질의어를 모두 제공한다. 검색 뿐만 아니라 삽입, 삭제, 수정할 수 있는 명령을 제공한다.
  - 뷰 정의
  - 트랜잭션 제어: 트랜잭션의 시작과 끝을 표시할 수 있다.
  - 임베디드 SQL과 동적 SQL: C, C++, Java와 같은 기존 언어에 SQL 문장을 포함하는 방법을 제공한다.
  - 무결성: SQL DDL은 무결성 제약 조건을 명시할 수 있는 방법을 제공한다.
  - 권한: SQL DDL은 데이터베이스 접근 권한을 명시할 수 있는 방법을 제공한다.
- 이 장에서 SQL를 설명하기 위해 사용하는 관계 스키마는 3장과 같다. 그림 4.1 참조.

4.2 기본 구조

- SQL 표현의 기본 구조는 select, from, where의 세 개의 절로 구성된다.
  - select: 관계 대수에서 추출 연산에 해당한다. 질의 결과에 나타나야 하는 속성의 목록을 나열하기 위해 사용된다.

Account-schema = (계좌번호, 지점명, 잔액)
Branch-schema = (지점명, 지점-도시, 자산)
Customer-schema = (고객명, 고객-거리, 고객-도시)
Depositor-schema = (고객명, 계좌번호)
Loan-schema = (대출번호, 지점명, 대출액)
Borrow-schema = (고객명, 대출번호)

<그림 4.1> 은행 데이터베이스의 스키마

  - from: 관계 대수에서 카르데시안 곱 연산에 해당한다. 질의를 평가하기 위해 검색해야 하는 관계를 나열하기 위해 사용된다.
  - where: 관계 대수에서 선택 조건에 해당한다. from 절에 나열한 관계의 속성에 관한 조건으로 구성된다.
- 전형적인 SQL 질의의 형태

$$\begin{aligned}&\text{select } A_1, A_2, \ldots, A_n\\&\text{from } r_1, r_2, \ldots, r_m\\&\text{where } p\end{aligned}$$

이 SQL 질의는 다음 관계 대수 표현식과 같다.

$$\Pi_{A_1,A_2,\ldots,A_n}(\sigma_P(r_1 \times r_2 \times \cdots \times r_m))$$

- SQL 질의의 결과는 관계이다.

4.2.1 select 절

- 예1) 대출이 있는 모든 지점의 이름을 찾아라.

```
select 지점명
from loan
```

관계는 집합 개념이므로 중복된 것은 관계에 등장할 수 없다. 그러나 SQL에서는 기본적으로 중복을 제거하지 않는다. 중복을 제거하고 싶으면 select 절 다음에 distinct을 사용해야 한다.

```
select distinct 지점명
from loan
```

- distinct의 반대는 all이지만 기본적으로 중복과 상관없이 모두 나열하므로 보통 all은 사용하지 않는다.
- 모든 속성을 전부 나열하고 싶으면 "*" 기호를 다음과 같이 사용할 수 있다.

```
select  loan.*
from    loan
```

위 SQL 문은 다음과 같이 보다 간단하게 작성할 수 있다.

```
select *
from loan
```

- select 절은 산술연산자, 속성이름, 상수를 사용한 산술표현식을 포함할 수 있다. (일반화 추출)

- 예2) 모든 계좌에 5% 이자를 지급한 결과를 보여라.

  select 계좌번호,잔액*1.05
  from *account*

  이 질의는 데이터베이스를 갱신하지 않는다.

4.2.2 where 절

- 예1) Perryridge 지점에서 대출된 대출 중에서 대출액이 1200보다 큰 모든 대출을 찾아라.

  select 대출번호
  from *loan*
  where 지점명='Perryridge' and
  대출액 $>$ 1200

- SQL에서는 문자열을 나타내기 위해 작은 따옴표('')을 사용한다.
- SQL은 관계 대수에서 사용한 수학적 기호 대신에 and, or, not을 사용한다.
- SQL은 비교 연산을 쉽게 나타낼 수 있도록 between 연산자를 제공한다.
- 예2) 대출액이 1000과 1500 사이에 있는 대출번호를 찾아라.

  select 대출번호
  from *loan*
  where 대출액 between 1000 and 1500

  위 SQL 문은 다음과 같다.

  select 대출번호
  from *loan*
  where 대출액 $>=$ 1000 and 대출액 $<=$ 1500

- SQL은 not between 연산자도 제공한다.

4.2.3 from 절

- 예1) 은행에 대출을 받은 모든 고객에 대해 그들의 이름, 대출번호, 대출액을 구하라.

  이 문제를 관계 대수로 표현하면 다음과 같다.

  $\Pi_{\text{고객명},borrower.\text{대출번호},\text{대출액}}(borrower \bowtie loan)$

  이것을 SQL 문으로 작성하면 다음과 같다.

  select 고객명,*borrower*.대출번호,대출액
  from *borrower*,*loan*
  where *borrower*.대출번호=*loan*.대출번호

- 속성 이름이 중복되어 모호성이 있는 경우에는 속성 이름 앞에 관계 이름을 붙인다.
- 예2) Perryridge 지점에서 대출을 받은 모든 고객에 대해 그들의 이름, 대출번호, 대출액을 구하라.

  select 고객명,*borrower*.대출번호,대출액
  from *borrower*,*loan*
  where *borrower*.대출번호=*loan*.대출번호 and
  지점명='Perryridge'

4.2.4 재명명 연산

- SQL은 관계와 속성을 재명명하는 메커니즘을 제공한다. 형태는 다음과 같다.

  옛-이름 as 새-이름

- as 구문은 select 또는 from 절에서 사용할 수 있다.
- 예)

  select 고객명,
  (신용한도 $-$ 사용금액) as 남은한도
  from *credit-info*

- 실제로 SQL에서는 빨셈 연산자와 혼동되므로 관계 이름이나 속성 이름에 '$-$' 기호를 사용할 수 없다.

4.2.5 투플 변수

- 투플 변수는 from 절에 as 구문을 이용하여 정의한다.
- 예1) 은행에 대출을 받은 모든 고객에 대해 그들의 이름, 대출번호, 대출액을 구하라.

  select 고객명, T.대출번호, 대출액
  from *borrower* as T, *loan* as S
  where T.대출번호 $=$ S.대출번호

- 투플 변수는 같은 관계에 있는 두 투플을 비교할 때 가장 유용하다.
- 예2) 자산이 적어도 Brooklyn 시에 위치하고 있는 한 지점의 자산보다 큰 모든 지점의 이름을 구하라.

  select distinct T.지점명
  from *branch* as T, *branch* as S
  where T.자산 $>$ S.자산 and
  S.지점-도시='Brooklyn'

4.2.6 문자열 연산

- 문자열 내에 작은 따옴표를 사용하고 싶으면 큰 따옴표를 이용하여 표현한다. 예) 'It"s right'
- like 연산자: 찾고 싶은 문자열 패턴을 지정할 때 사용한다. 문자열 패턴은 다음 두 기호를 이용하여 지정한다.
  - '%': 어떤 부분 문자열과도 일치
  - '_': 어떤 한 문자와도 일치
- 예1) 'Perry%': "Perry"로 시작하는 모든 문자열
- 예2) '___': 정확하게 세 개의 문자로 이루어진 모든 문자열

- 예3) 고객 거리 이름에 “Main”이라는 부분 문자열을 포함하는 모든 고객의 이름을 찾아라.

```
select 고객명
from customer
where 고객-거리 like '%Main%'
```

- 문자열 내에 '%'와 '_'을 나타내기 위해 escape 구문을 사용할 수 있다.

- 예4) like 'ab\%cd%' escape '\': “ab%cd”로 시작되는 모든 문자열

4.2.7 출력되는 투플의 순서

- SQL 문의 실행 결과는 order by 절을 이용하여 출력의 순서를 정렬할 수 있다.

- 예1) 고객 이름을 기준으로 정렬

```
select distinct 고객명
from borrower,loan
where borrower.대출번호=loan.대출번호
      and 지점명='Perryridge'
order by 고객명
```

order by는 기본적으로 오름차순으로 항목을 정렬한다.

- 오름차순과 내림차순은 asc 또는 desc로 나타낸다.

- 예2) 대출액을 기준으로 내림차순으로 정렬하되, 대출액이 같으면 대출번호를 기준으로 오름차순으로 정렬하여라.

```
select *
from loan
order by 대출액 desc,대출번호 asc
```

4.3 중복

- 중복을 허용하였을 때 SQL 질의의 의미를 관계연산자의 다중집합(multiset) 버전으로 다음과 같이 정의할 수 있다. 여기서 r_1과 r_2는 다중집합이다.
  - r_1에 투플 t_1의 사본이 c_1개 있고, t_1이 σ_θ를 만족하면 $\sigma_\theta(r_1)$에는 t_1의 사본이 c_1개 있다.
  - r_1에 있는 t_1의 각 사본에 대해, $\Pi_A(t_1)$이 단일 투플 t_1에 대한 추출을 나타낸다면 $\Pi_A(r_1)$에는 $\Pi_A(t_1)$의 사본이 있다.
  - r_1에 투플 t_1의 사본이 c_1개 있고, r_2에 투플 t_2의 사본이 c_2개 있으면 $r_1 \times r_2$에는 $t_1.t_2$의 사본이 총 $c_1 * c_2$개 있다.

4.4 집합 연산

4.4.1 합집합 연산

- 예) 은행에 계좌 또는 대출이 있는 모든 고객의 이름을 찾아라.

```
(select 고객명 from depositor)
union
(select 고객명 from borrower)
```

- union은 자동으로 중복을 제거한다. 중복을 제거하고 싶지 않으면 union 대신에 union all을 사용한다.

4.4.2 교집합 연산

- 예) 은행에 계좌와 대출을 모두 가지고 있는 모든 고객의 이름을 찾아라.

```
(select 고객명 from depositor)
intersect
(select 고객명 from borrower)
```

- intersect도 union과 마찬가지로 자동으로 중복을 제거한다. 중복을 제거하고 싶지 않으면 intersect 대신에 intersect all을 사용한다.

4.4.3 차집합 연산

- 예) 은행에 계좌만 있고 대출은 없는 모든 고객의 이름을 찾아라.

```
(select 고객명 from depositor)
except
(select 고객명 from borrower)
```

- except도 자동으로 중복을 제거한다. 중복을 제거하고 싶지 않으면 except 대신에 except all을 사용한다.

4.5 집계 함수

- SQL은 다음과 같은 집계 함수를 제공한다.
  - 평균: avg
  - 최소: min
  - 최대: max
  - 총합: sum
  - 개수: count

- 예1) Perryridge 지점의 계좌의 평균 잔액을 구하라.

```
select avg(잔액)
from account
where 지점명='Perryridge'
```

- group by 절을 이용하여 그룹별로 집계 함수를 적용할 수 있다.

- 예2) 각 지점의 계좌의 평균 잔액을 구하라.

```
select 지점명,avg(잔액)
from account
group by 지점명
```

- 집계 함수를 적용하기 전에 중복를 제거할 필요가 있을 수 있다. 이 때에는 distinct 키워드를 사용한다.

- 예3) 각 지점의 계좌를 가진 고객의 수를 계산하라.

```
select 지점명,count(distinct 고객명)
from depositor,account
where depositor.계좌번호=account.계좌번호
group by 지점명
```

- 그룹별로 집계를 계산할 때 having 절을 이용하여 결과를 제한할 수 있다.

- 예4) 지점의 계좌의 평균 잔액이 1200 이상인 지점의 지점명과 평균 잔액을 구하라.

```
select 지점명,avg(잔액)
from account
group by 지점명
having avg(잔액) > 1200
```

- 예) 모든 계좌의 평균 잔액을 구하라.

```
select avg(잔액)
from account
```

- 어떤 관계의 총 투플 수를 계산하고 싶으면 다음과 같이 한다.

```
select count(*)
from customer
```

- 예5) Harrison 시에 거주하면서 세 개 이상의 계좌를 가진 모든 고객의 평균 잔액을 구하라.

```
select depositor.고객명,avg(잔액)
from depositor,account,customer
where depositor.계좌번호=account.계좌번호
    and depositor.고객명=customer.고객명
    and customer.고객-도시='Harrison'
group by depositor.고객명
having count(distinct depositor.계좌번호) >= 3
```

SQL 문에서 where 절과 having 절이 같이 사용되면 SQL은 먼저 where 절을 먼저 적용한다. 따라서 위 SQL 문에서는 where 절에 만족하는 투플들을 먼저 그룹핑한 다음 having 절을 적용하여 만족하지 않는 그룹은 버린다.

4.6 널 값

- null 키워드를 이용하여 특정 속성의 값이 널인지 비교할 수 있다.

- 예) 대출액이 널 값으로 되어 있는 대출의 대출번호를 찾아라.

```
select 대출번호
from loan
where 대출액 is null
```

- is not null을 이용하여 특정 속성의 값이 널이 아닌지 비교할 수 있다.

- SQL에서 널 값의 처리는 관계 대수에서 널 값의 처리와 같다.

4.7 중첩 하위 질의식

4.7.1 집합 멤버쉽

- in 연결자를 이용하여 집합 멤버쉽을 검사할 수 있다.

- 예1) 은행에 계좌와 대출을 모두 가지고 있는 모든 고객의 이름을 찾아라.

```
select distinct 고객명
from borrower
where 고객명 in
    (select 고객명 from depositor)
```

- 예2) Perryridge 지점에 계좌와 대출을 모두 가지고 있는 모든 고객의 이름을 찾아라.

```
select distinct 고객명
from borrower,loan
where borrower.대출번호=loan.대출번호
        and 지점명='Perryridge'
        and (지점명,고객명) in
            (select 지점명,고객명
            from depositor,account
            where depositor.계좌번호=
                account.계좌번호)
```

- 예3) 은행에 계좌는 있지만 대출은 없는 모든 고객의 이름을 찾아라.

```
select distinct 고객명
from borrower
where 고객명 not in
        (select 고객명 from depositor)
```

- in과 not in은 열거형 집합에 적용할 수 있다.

- 예4) 은행에 대출이 있지만 이름이 Smith 또는 Jones이 아닌 모든 고객의 이름을 찾아라.

```
select distinct 고객명
from borrower
where 고객명 not in ('Smith','Jones')
```

4.7.2 집합 비교

- "집합 내에 자신보다 작은 값이 최소한 하나 존재"는 SQL에서 $>$ some으로 표현한다.
- 예1) 자산이 적어도 Brooklyn 시에 위치하고 있는 한 지점의 자산보다 큰 모든 지점의 이름을 구하라.

```
select 지점명
from branch
where 자산 > some
        (select 자산
        from branch
        where 지점-도시='Brooklyn')
```

- $<$ some, $<=$ some, $>=$ some, $=$ some, $<>$ some 등도 가능하다.
- $=$ some은 in과 등가이지만 $<>$ some과 not in은 다르다.
- 예2) 자산이 Brooklyn 시에 위치하고 있는 모든 지점의 자산보다 큰 모든 지점의 이름을 구하라.

```
select 지점명
from branch
where 자산 > all
        (select 자산
        from branch
        where 지점-도시='Brooklyn')
```

- $<$ all, $<=$ all, $>=$ all, $=$ all, $<>$ all 등도 가능하다.
- $<>$ all은 not in과 등가이다.
- 예3) 계좌들의 평균 잔액이 가장 많은 지점을 찾아라.

```
select 지점명
from account
group by 지점명
having avg(잔액) >= all
        (select avg(잔액)
        from account
        group by 지점명)
```

4.7.3 빈 관계에 대한 검사

- exists 구문을 이용하여 하위 질의 결과가 투플을 가지는지 아닌지를 검사할 수 있다.
- exists A: 관계 A가 투플을 가지면 참이고, 가지지 않으면 거짓이다.
- 예1) 은행에 계좌와 대출을 모두 가지고 있는 모든 고객의 이름을 찾아라.

```
select 고객명
from borrower
where exists
    (select * from depositor
    where depositor.고객명=borrower.고객명)
```

- 관계 A가 관계 B를 포함하는 여부는 "not exists (B except A)"를 이용할 수 있다.
- 예2) Brooklyn 시에 위치한 모든 지점에 계좌를 가지고 있는 고객의 이름을 찾아라.

```
select distinct S.고객명
from depositor as S
where not exists
        (select 지점명
        from branch
        where 지점-도시='Brooklyn')
        except
        (select R.지점명
        from depositor as T, account as R
        where T.계좌번호 = R.계좌번호
            and S.고객명 = R.고객명)
```

- 하위질의에 선언된 투플 변수는 상위질의에서 사용될 수 없지만 상위질의에 선언된 투플 변수는 하위질의에 사용될 수 있다.

4.7.4 중복 투플 부재에 대한 검사

- 중복된 투플이 있는지 검사할 때에는 unique 구문을 사용한다.
- 예1) Perryridge 지점에 하나의 계좌만을 가진 모든 고객을 찾아라.

```
select T.고객명
from depositor as T
where unique
        (select R.고객명
        from account, depositor as R
        where T.고객명 = R.고객명
            and R.계좌번호 = account.계좌번호
            and account.지점명='Perryridge')
```

- 예2) Perryridge 지점에 두 개 이상의 계좌를 가진 모든 고객을 찾아라.

```
select T.고객명
from depositor as T
where not unique
        (select R.고객명
        from account, depositor as R
        where T.고객명 = R.고객명
            and R.계좌번호 = account.계좌번호
            and account.지점명='Perryridge')
```

4.8 뷰

- 뷰는 **create view** 문장을 이용하여 정의한다.
- **create view** 문장의 형식

create view v **as** $\langle$query expression$\rangle$

여기서 v는 뷰의 이름이다.

- 예) 지점과 고객명으로 구성된 뷰

```
create view all-customer as
    (select 지점명, 고객명
    from depositor, account
    where depositor.계좌번호 = account.계좌번호)
  union
    (select 지점명, 고객명
    from borrower, account
    where borrower.계좌번호 = account.계좌번호)
```

- 뷰의 속성 이름을 새롭게 정의할 수 있다.

```
create view
   branch-total-loan(지점명,전체대출액) as
      (select 지점명, sum(대출액)
      from loan
      group by 지점명)
```

4.9 복합 질의

4.9.1 유도된 관계

- 예1) 지점의 계좌들의 평균 잔액이 1200보다 큰 지점들의 평균 잔액을 구하라.

```
select 지점명,잔액평균
from (select 지점명, avg(잔액)
        from account
        group by 지점명)
        as (branch-avg(지점명,잔액평균))
where 잔액평균 > 1200
```

이 예는 앞서 having 절을 이용하여 구하였다.

- 예2) 각 지점의 계좌의 총잔액이 가장 큰 지점의 총잔액을 구하라.

```
select max(총잔액)
from (select 지점명, sum(잔액)
        from account
        group by 지점명)
        as (branch-total(지점명,총잔액))
```

4.9.2 with 절

- with 절을 이용하여 일시적인 뷰를 만들 수 있다. with 절을 이용하여 만든 뷰는 그 질의에서만 유효한 일회성 뷰이다.

- 예1) 최대잔액을 가진 계좌의 계좌번호를 찾아라.

```
with max-balance(최대잔액) as
    select max(잔액)
    from account
select 계좌번호
from account, max-balance
where account.잔액 = max-balance.최대잔액)
```

- with 절은 SQL:1999에서 처음 도입되었으며, 현재 소수의 데이터베이스만 이를 지원한다.

- 예2) 모든 지점의 총잔액의 평균보다 지점의 총잔액이 적은 지점을 찾아라.

```
with branch-total(지점명,총잔액) as
    select 지점명,sum(잔액)
    from account
    group by 지점명
with branch-total-avg(총잔액평균) as
    select avg(총잔액)
    from branch-total
select 지점명
from branch-total, branch-total-avg
where branch-total.총잔액 >=
        branch-total-avg.총잔액평균)
```

4.10 데이터베이스의 수정

4.10.1 삭제

- 삭제는 SQL에서 다음과 같이 표현한다.

```
delete from r
where P
```

여기서 r은 관계이며, P는 조건이다.

- 예1) Perryridge 지점에 있는 모든 계좌를 삭제하라.

```
delete from account
where 지점명='Perryridge'
```

- 예2) 대출액이 1300에서 1500 사이에 있는 모든 대출을 삭제하라.

```
delete from loan
where 대출액 between 1300 and 1500
```

- 예3) Needham 시에 위치한 모든 지점에 있는 계좌를 삭제하라.

```
delete from account
where 지점명 in
        (select 지점명
        from branch
        where 지점-도시='Needham')
```

- 예4) 은행의 평균 잔액보다 적은 계좌를 모두 삭제하라.

```
delete from account
where 잔액 <
        (select avg(잔액)
        from account)
```

검사를 먼저 한 다음에 삭제하는 것이 중요하다.

4.10.2 삽입

- 예) Smith가 Perryridge 지점에 계좌번호가 A-973이고 잔액이 1200인 계좌를 가지고 있다는 정보를 데이터베이스에 추가해라.

```
insert into account
    values ('A-973','Perryridge',1200)
insert into depositor
    values ('Smith','A-973')
```

4.10.3 갱신

- 예1) 모든 계좌에 대해 5%의 이자를 지급해라.

```
update account
set 잔액=잔액*1.05
```

- 예2) 잔액이 1000 이상인 계좌에 대해서만 5%의 이자를 지급해라.

```
update account
set 잔액=잔액*1.05
where 잔액 >= 1000
```

- update 문에서 사용하는 where 절은 select 문의 where 절과 같다.

- 예3) 잔액이 10000 보다 큰 계좌에 대해서는 6%의 이자를 지급하고, 그 이하의 계좌에 대해서는 5%의 이자를 지급하라.

```
update account
set 잔액=잔액*1.06
where 잔액 > 10000

update account
set 잔액=잔액*1.05
where 잔액 <= 10000
```

  여기서는 두 update 문의 실행 순서가 중요하다.

- 갱신의 실행 순서 문제를 해결하기 위해 SQL은 case 구문을 제공한다. 이 구문을 이용하면 다른 종류의 갱신을 하나의 SQL 문에서 할 수 있다.

```
update account
set 잔액=
    case
        when 잔액 <= 10000 then 잔액*1.05
        else 잔액*1.06
    end
```

- case 구문의 일반적인 형태

```
case
        when pred_1 then result_1
        when pred_2 then result_2
        ...
        when pred_n then result_n
        else result_0
end
```

4.10.4 뷰 갱신

- SQL에서는 뷰 갱신 문제를 극복하기 위해 하나의 관계를 이용하여 정의된 뷰에 대해서만 갱신할 수 있도록 해준다. 이 때 모르는 값은 null 값으로 저장된다.

4.10.5 트랜잭션

- SQL에서 트랜잭션은 일련의 질의 또는 갱신 문장으로 구성된다.

- 트랜잭션의 시작은 명백하게 나타내지 않지만 끝은 다음 두 문장 중 하나로 나타낸다.
  - commit work: 지금까지 모든 갱신을 데이터베이스에 영구적으로 반영한다.
  - rollback work: 지금까지 모든 갱신을 취소하고 트랜잭션이 시작하기 전 상태로 복원한다.

- 이런 문장을 실행하지 않고 프로그램을 종료하면 데이터베이스에 따라 두 문장 중 하나를 실행하고 끝낸다.

4.11 데이터 정의 언어

4.11.1 SQL에서 제공하는 도메인 타입의 종류

- char(n): 고정된 크기(n)의 문자열
- varchar(n): 최대 크기가 n인 가변적 문자열
- int: 정수형
- smallint
- numeric(p,d): 사용자 지정 정밀도를 가지는 부동소수점 타입으로 수는 전체 p 숫자로 구성되어 있고, 정밀도는 d이다.
- real, double precision: float보다는 두 배의 정밀도를 가지는 부동소수점 타입
- float(n): 정밀도가 최소 n인 부동소수점 타입
- date: 날짜형(년도,월,일)
  - 명시하는 방법: '2001-04-05'
  - 년도, 월, 일은 extract(*field* from d) 구문을 이용하여 date 타입으로부터 추출할 수 있다. 이 때 *field*는 year, month, day 중 하나를 사용하면 된다.
- time: 시간형(시,분,초)
  - 명시하는 방법: '09:30:25'
  - 시, 분, 초도 date 타입에서 사용하는 구문과 같은 구문을 이용하여 추출할 수 있다. 이 때 *field*는 hour, minute, second 중 하나를 사용하면 된다.
- timestamp: date와 time이 결합된 타입
  - 명시하는 방법: '2001-04-05 09:30:25'
- 문자열 타입을 다른 타입으로 바꿀 때 사용하는 구문: cast e as t
- 문자열 타입을 다른 타입으로 바꾸기 위해서는 문자열이 바뀔 타입의 형태를 갖추고 있어야 한다.

- SQL은 모든 도메인에 비교 연산을 적용할 수 있으며, 모든 정수와 부동소수점 타입들에 산술 연산을 적용할 수 있다.
- date 타입 간에 그리고 time 타입 간에는 '$-$' 연산을 적용할 수 있으며 결과는 interval 타입이다.
- 호환 타입(compatible type) 간에는 비교 연산을 적용할 수 있다.
- null은 모든 타입의 도메인에 속한다.

4.11.2 스키마 정의

- 스키마는 create table 명령을 이용하여 정의한다.

$$\begin{aligned}\text{create table}\quad & r(A_1D_1, A_2D_2, \ldots, A_nD_n,\\ & \langle \text{무결성 제약조건}_1 \rangle,\\ & \ldots,\\ & \langle \text{무결성 제약조건}_k \rangle)\end{aligned}$$

여기서 r은 관계의 이름이고, A_i는 속성의 이름이다. D_i는 A_i 속성의 도메인이다.

- 가능한 무결성 제약조건은 다음과 같다.
  - primary key$(A_{j1}, A_{j2}, \ldots, A_{jm})$: 속성 A_{j1}, A_{j2}, ..., A_{jm}이 관계의 주키가 된다. 주키는 null이 될 수 없으며, 독특하여야 한다.
  - check(P): 관계에 있는 모든 투플이 만족해야 하는 조건을 명시하기 위해 사용된다.

  이 외에 다른 제약조건도 있지만 그것에 대해서는 6장에서 다룬다.

- 예) *account* 관계의 스키마 정의

```
create table account
    (계좌번호 char(10),
    지점명 char(15),
    잔액 int,
    primary key(계좌번호),
    check(잔액 >= 0))
```

- 예) 학생 관계의 스키마 정의

```
create table student
    (성명 char(15) not null,
    학번 char(10),
    과정 char(15),
    primary key(학번),
    check(과정 in ('학부','석사','박사')))
```

스키마에서 속성을 정의할 때 not null를 명시하면 그 속성은 null 값을 가질 수 없다. 주키는 명시하지 않아도 기본적으로 null 값을 가질 수 없다.

- unique$(A_{j1}, A_{j2}, \ldots, A_{jm})$ 구문을 이용하여 후보키를 지정할 수 있다.
- 관계를 데이터베이스에서 삭제할 때에는 drop table 명령을 사용한다.

$$\text{drop table } r$$

- 관계는 그대로 데이터베이스에 두고, 그 관계에 있는 모든 투플만 삭제할 때에는 다음 명령을 사용한다.

$$\text{delete from } r$$

- alter table 명령을 이용하여 기존 관계의 스키마를 변경할 수 있다. 속성의 추가는 다음과 같으며,

$$\text{alter table } r \text{ add } A\ D$$

속성의 삭제는 다음과 같다.

$$\text{alter table } r \text{ drop } A$$

4.12 임베디드 SQL

- SQL은 강력한 선언적 질의 언어를 제공하지만 일반 프로그래밍 언어 내에서 SQL을 사용할 필요성이 있다.
  - 모든 질의를 SQL로 표현할 수 있는 것은 아니기 때문에 범용 프로그래밍 언어의 표현력을 이용할 필요가 종종 있다.
  - 질의 결과를 그래픽 인터페이스로 표현하는 기능 등을 제공하기 위해서는 프로그래밍 언어을 사용할 수 밖에 없다.
- SQL를 내장한 일반 프로그램은 컴파일하기 전에 전처리가 필요하다. 이 전처리에서는 SQL 문장을 호스트 언어 선언과 함수 호출로 바꾸어준다.
- 보통 다음과 같은 구문을 이용하여 SQL을 범용 프로그래밍 언어에 내장한다.

$$\text{EXEC SQL } < \text{SQL 문장} > \text{ END-EXEC}$$

자바는 다음과 같은 구문을 사용한다.

$$\text{\#SQL}\{ < \text{SQL 문장} > \};$$

- 예) 호스트 언어에 *amount*라는 변수가 있다고 하자. 그러면 다음과 같은 질의를 호스트 언어에 내장할 수 있다.

```
EXEC SQL
    declare c cursor for
    select 고객명,고객-거리
    from depositor,customer,account
    where depositor.고객명=customer.고객명 and
          account.계좌번호=depositor.계좌번호 and
          잔액 > :amount
END-EXEC
```

여기서 변수 c는 이 질의의 cursor라 하고, 호스트 언어의 변수를 SQL 문장 내에서 사용할 때에는

변수 앞에 ':'를 첨가한다. 이 질의를 실제 실행하기 위해서는 open 문을 이용해야 한다.

EXEC SQL open *c* END-EXEC

질의 결과는 임시 관계에 저장된다. 질의 결과의 투플은 fetch 문을 이용하여 얻을 수 있다. 위 질의의 결과 관계는 두 개의 속성으로 구성되므로 두 개의 변수(*cn*, *cc*)가 필요하다.

EXEC SQL fetch *c* into :*cn*,:*cc* END-EXEC

fetch 문은 하나의 투플만을 반환해주므로 모든 투플에 대해 작업을 하기 위해서는 루프를 이용해야 한다. 루프의 끝은 SQL communication-area (SQLCA) 변수를 통해 알 수 있다. 질의 결과가 더 이상 필요없으면 close 문을 이용하여 삭제하여야 한다.

EXEC SQL close *c* END-EXEC

- 데이터베이스 변경 명령은 질의와 달리 새로운 관계를 결과로 주지 않기 때문에 내장하기가 더 쉽다.

```
EXEC SQL
    declare c cursor for
    select *
    from account
    where 지점명='Perryridge'
    for update
END-EXEC
```

갱신에 사용할 질의를 먼저 내장한 다음에 open 문을 이용하여 질의를 실행한다. 그 다음 fetch 문과 update를 이용하여 실제 갱신을 한다.

```
EXEC SQL
    fetch c into :an,:bn,:ab;
    update account
    set 잔액=잔액+100
    where current of c;
END-EXEC
```