3. SQL

DATABASE

Contents

1. 배경

Ch 3. SQL

역사

- SQL의 최초 버전
  - IBM, San Jose Research Laboratory (지금은 Almaden Research Center)
  - 1970년대 초반, System R 프로젝트의 일부분으로 Sequel이라는 언어를 구현
  - Sequel 언어는 계속 발전되다가, SQL(Structured Query Language)로 이름이 바뀜
  - 표준 관계형 데이터베이스 언어가 됨

- SQL-86
  - ANSI(American National Standard Institute)와 ISO(International Organization for Standard)는 SQL 표준을 출간
  - IBM은 SAA-SQL(Systems Application Architecture Database Interface)라는 자체적인 기업 SQL 표준을 발행

- SQL-89

- SQL-92

- SQL:1999 (Ch. 9)

SQL 언어의 구성

- 데이터 정의 언어(DDL: Data Definition Language)
- 데이터 조작 언어(DML: Data Manipulation Language)
- 무결성(integrity)
- 뷰 정의(view definition)
- 트랜잭션 제어(transaction control)
- 내장 SQL (embedded SQL)과 동적 SQL (dynamic SQL)
- 인증(authorization)

예제에서 사용하는 스키마

branch(*branch_name*, *branch_city*, *assets*)

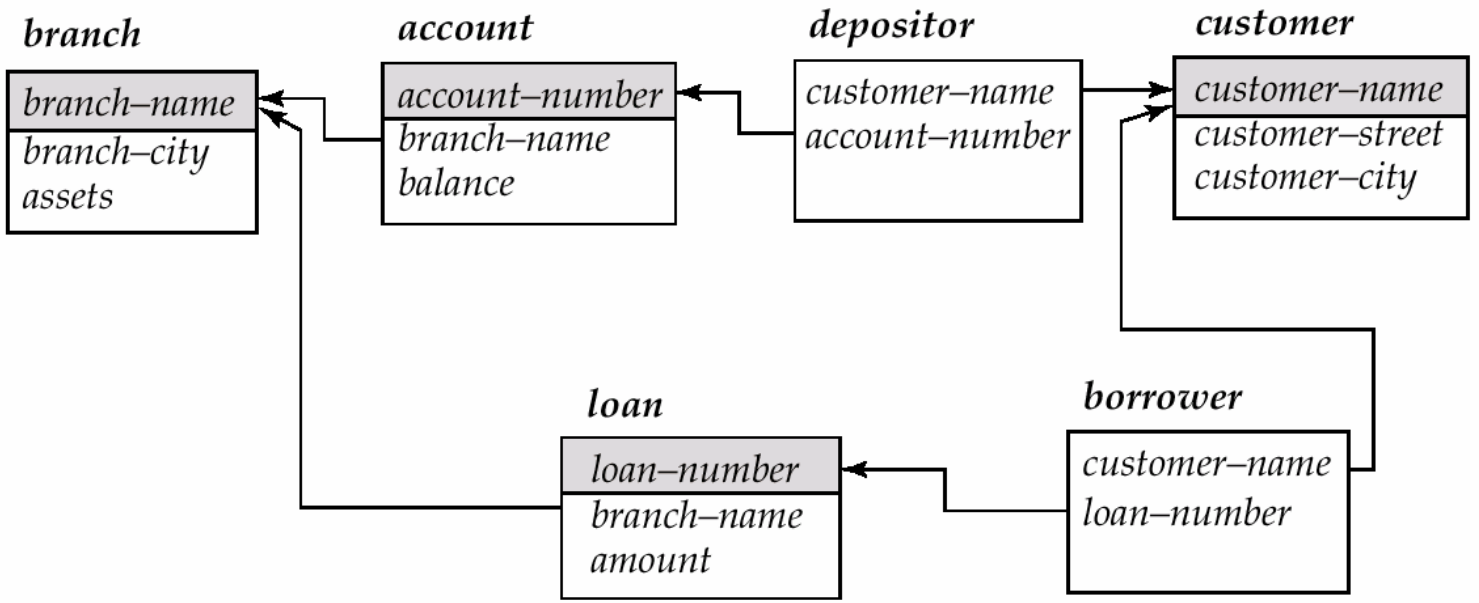
customer(*customer_name*, *customer_street*, *customer_city*)

loan(*loan_number*, *branch_name*, *amount*)

borrower(*customer_name*, *loan_number*)

account(*account_number*, *branch_name*, *balance*)

depositor(*customer_name*, *account_number*)



2. 데이터 정의 언어(DDL)

Ch 3. SQL

데이터 정의 언어 (DDL)

- DDL(Data Definition Language)은 릴레이션의 집합 뿐만 아니라 다음과 같은 각 릴레이션에 대한 정보도 포함
  - 각 릴레이션의 스키마
  - 각 속성들과 관련된 값들의 도메인
  - 무결성 제약조건(integrity constraints)
  - 각 릴레이션에서 유지되어야 할 인덱스들의 집합
  - 각 릴레이션의 보안과 권한 정보
  - 각 릴레이션의 디스크에서의 물리적인 저장 구조

기본 도메인 타입

- **char(n).** 사용자가 지정하는 길이 *n* 을 갖는 고정 길이 문자열
- **varchar(n).** 사용자가 지정하는 최대 길이 *n* 을 갖는 가변 길이 문자열
- **int.** 정수 (기계 종속적인 정수의 유한 부분 집합)
- **smallint.** 작은 정수 (정수 도메인 타입의 기계 종속적인 부분 집합)
- **numeric(p,d).** 사용자가 지정하는 정확도(precision)를 갖는 고정점 수 (fixed point number). *p* 개의 숫자에서 *d* 개는 소수점의 오른쪽에 있음
- **real, double precision.** 기계 종속적인 정확도를 가지는 부동 소수와 두 배의 정확도를 지닌 부동 소수
- **float(n).** 적어도 *n* 개의 숫자의 정확도를 가지는 부동 소수

Create Table 명령

- SQL 릴레이션은 **create table** 명령을 사용하여 정의

  create table *r* (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *r* 은 릴레이션의 이름
  - 각각의 A_i 은 릴레이션 *r* 스키마의 속성명
  - D_i 은 속성 A_i의 도메인 안의 값의 데이터 타입

- 예제:

```
create table branch
     (branch_name      char(15) not null,
      branch_city      char(30),
      assets           integer)
```

은행 데이터베이스의 일부분에 대한 SQL 데이터 정의

```
create table customer
    (customer-name    char(20),
     customer-street  char(30),
     customer-city    char(30),
     primary key (customer-name))

create table branch
    (branch-name      char(15),
     branch-city      char(30),
     assets           integer,
     primary key (branch-name),
     check (assets >= 0))

create table account
    (account-number   char(10),
     branch-name      char(15),
     balance          integer,
     primary key (account-number),
     check (balance >= 0))

create table depositor
    (customer-name    char(20),
     account-number   char(10),
     primary key (customer-name, account-number))
```

Drop과 Alter Table 명령

- **drop table** 명령은 데이터베이스로부터 제거된 릴레이션에 대한 모든 정보를 삭제

- **alter table** 명령
  - 이미 존재하는 릴레이션에 속성을 추가하기 위해 사용함. 릴레이션의 모든 투플에 대해서 새로운 속성을 위한 값으로 널이 할당됨.
  - **alter table** 명령의 형식

    ```
    alter table r add A D
    ```

    r은 이미 존재하는 릴레이션의 이름이고, *A*는 추가될 속성의 이름, *D* 는 그 속성의 도메인임
  - 릴레이션의 속성을 삭제하는데 사용할 수 있음

    ```
    alter table r drop A
    ```

    *A*는 릴레이션의 이름, r의 속성 이름임
  - 속성의 삭제는 많은 데이터베이스에서는 지원되지 않음

3. 기본 구조

Ch 3. SQL

기본 구조

- 관계형 데이터베이스는 각각이 유일한 이름으로 명명된 릴레이션들의 집합으로 구성되어져 있음
- 전형적인 SQL 질의는 다음과 같은 형식을 가짐

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  where P

  - A_i 는 속성을 나타냄
  - r_i 는 릴레이션을 나타냄
  - P 는 술어를 나타냄

- 질의어는 관계 대수 표현식으로 동등하게 표현될 수 있음

  $\Pi_{A1, A2, ..., An}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$

- SQL 질의어의 결과는 릴레이션

select 절 (1)

- select 절은 관계 대수의 추출(projection) 연산과 같은 의미임. 질의어의 결과로 원하던 속성의 목록이 나옴
- loan 릴레이션에서 모든 지점의 이름을 찾아라

```
select branch_name
from loan
```

- 위의 질의어는 순수 관계 대수 문법으로 다음과 같이 표현

$$\Pi_{\text{branch\_name}}(loan)$$

- select절의 별표 기호(*)는 “모든 속성들”를 뜻함

```
select *
from loan
```

select 절 (2)

- SQL은 질의어의 결과뿐만 아니라, 릴레이션에서도 중복을 허용
- 중복의 제거를 위해서는, select 뒤에 distinct라는 키워드를 삽입
- loan 릴레이션에서 모든 지점의 이름을 찾되, 중복은 제거하라

```
select distinct branch_name
from loan
```

- 중복을 제거하지 않는 것에 대해 명시하기 위해 all이라는 키워드를 사용

```
select all branch_name
from loan
```

select 절 (3)

- select 절은 또한, 상수나 투플의 속성에 적용되는 +, -, *, / 연산자를 포함하는 산술적인 표현들을 사용할 수 있음

```
select loan_number, branch_name, amount * 100
from loan
```

where 절 (1)

- where 절은 관계 대수의 선택(selection) 술어와 대응
  - from 절에서 나타나는 릴레이션의 속성을 포함하는 술어로 구성

- Perryridge 지점에서 $1200 이상의 금액을 대출한 모든 대출번호를 찾아라

```
select loan_number
from loan
where branch_name = 'Perryridge' and amount > 1200
```

  - SQL은 where 절에 ^, ∨, ¬ 와 같은 수학적인 기호를 사용하는 대신 **and, or, not**과 같은 논리적인 접속사를 사용

where 절 (2)

- **between** 비교 연산자

- Ex) 대출한 금액이 $90,000와 $100,000 사이인 대출 번호를 찾아라 (즉, ≥$90,000이고 ≤$100,000)

```
select loan_number
from loan
where amount between 90000 and 100000
```

from 절

- **from** 절은 관계 대수의 카티션곱에 해당
- 질의문에서 찾아보기를 원하는 릴레이션을 나열
- borrower x loan의 카티션곱을 구하라

```
select *
from borrower, loan
```

- Perryridge 지점에 대출을 받고 있는 모든 고객의 이름, 대출 번호, 대출 액을 구하라

```
select customer_name, borrower.loan_number, amount
from   borrower, loan
where  borrower.loan_number = loan.loan_number
    and branch_name = 'Perryridge'
```

rename 연산

- SQL은 릴레이션과 속성의 이름을 재명명(renaming)하는데, **as** 절을 사용
  - `old-name` **as** `new-name`

- 모든 고객의 이름, 대출 번호, 대출액을 구하라. 단, loan-number를 loan-id로 바꿔라

```
select customer_name, borrower.loan_number as loan_id,
   amount
from borrower, loan
where borrower.loan_number = loan.loan_number
```

투플 변수

- 투플 변수는 **from** 절에서 **as** 절을 통해 정의

- 은행에 대출을 지닌 모든 고객에 대해 그들의 이름과, 대출번호와 대출 총액을 구하라

```
select customer_name, T.loannumber, S.amount
from borrower as T, loan as S
where  T.loan_number = S.loan_number
```

- Brooklyn에 위치하고 있는 적어도 한 지점보다는 큰 자산 총액을 가지는 모든 지점들의 이름을 구하라

```
select distinct T.branch_name
from branch as T, branch as S
where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

문자열 연산

- SQL에서는 문자열의 비교를 위해서 다음과 같은 두 개의 특수 문자를 사용하여 나타낼 수 있음
  - % : % 문자는 어떠한 부분 문자열과도 일치한다
  - _ : _문자는 어떠한 한 문자와도 일치한다

- 이름에 “Main”이라는 부분문자열이 포함된 거리에 살고 있는 모든 고객들의 이름을 구하라

```
select customer_name
from customer
where customer_street like '%Main%'
```

- “Main%”로 시작하는 모든 문자열과 일치

```
like 'Main\%' escape  '\'
```

- SQL은 다음과 같은 다양한 문자열을 위한 함수를 제공
  - 문자열 연결(concatenation) - “||”를 사용
  - 대문자 소문자 간의 변환
  - 문자열의 길이 구하기, 부분 문자열 추출 등

투플 출력의 순서

- Perryridge 지점의 대출을 가진 모든 고객들을 알파벳 순서로 나열하라

```
    select distinct customer_name
    from  borrower, loan
    where borrower.loan_number = loan.loan_number
and
          branch_name = 'Perryridge'
    order by customer_name
```

- 정렬 순서를 명시하기 위해서, 내림차순을 위해서는 **desc**를, 오름차순을 위해서는 **asc**를 명시할 수 있음. 기본적으로, 오름차순으로 항목을 나열
  - E.g. **order by** *customer_name* **desc**

중복 (1)

- SQL은 질의의 결과에 어떤 투플이 있는지 뿐만 아니라, 결과에 그러한 투플들이 각각 얼마의 사본을 가지는지도 형식적으로 정의

- 중복의 의미를 관계 대수 연산의 다중집합(multiset) 버전을 사용하여 정의. 주어진 다중집합 릴레이션 r_1과 r_2에 대해

  1. r_1에 t_1 투플의 사본이 c_1개 있고, t_1이 선택 σ_θ를 만족한다면, $\sigma_\theta(r_1)$ 에는 t_1 의 c_1 개의 사본이 있음
  2. r_1 에 있는 t_1 의 각각의 사본에 대해, $\Pi_A(t_1)$ 이 단일 투플 t_1에 대한 추출을 나타낸다면, $\Pi_A(r_1)$에는 투플 $\Pi_A(t_1)$의 사본이 있음
  3. r_1 에는 투플 t_1의 사본이 c_1개 있고, r_2에는 투플 t_2의 사본이 c_2 개 있다면, $r_1 \times r_2$ 에는 투플 $t_1 . t_2$ 의 사본이 $c_1 \times c_2$ 개 있음

중복 (2)

- 예제 : 스키마 (A,B)를 가지는 릴레이션 r_1과 스키마 (C)를 가지는 r_2 가 다음과 같은 다중 집합이라고 하자
  - r_1 = {(1,*a*) (2,*a*)} r_2 = {(2), (3), (3)}

- 그러면 $\Pi_B(r_1)$ x r_2 가 {(*a*,2), (*a*,2), (*a*,3), (*a*,3), (*a*,3), (*a*,3)} 임에 반하여, $\Pi_B(r_1)$는 {(a), (a)}이다

- SQL에서 다음과 같이 표현하고,

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

이를 다중 집합 버전을 사용하여 관계 대수 표현식으로 표현하면 다음과 같이 나타낼 수 있다

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

4. 집합 연산

Ch 3. SQL

집합 연산 (1)

- SQL 연산 **union, intersect, except**는 릴레이션에 대해 각각 관계 대수 연산 ∪, ∩, – 와 같은 연산을 수행

- 위의 연산들 각각은 자동적으로 중복을 제거. 만약 모든 중복을 유지하고 싶다면 각각의 것 대신에 **union all, intersect all, except all**을 사용

- r에서 m번, s에서 n번 중복된 투플이 있다고 가정할 때, 결과에서 중복된 투플의 개수는 다음과 같음
  - *r* **union all** *s* : $m + n$
  - *r* **intersect all** *s* : $\min(m, n)$
  - *r* **except all** *s* : $\max(0, m - n)$

집합 연산 (2)

- 계좌나 대출 혹은 둘 다 가지고 있는 모든 고객의 이름을 구하라

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

- 계좌와 대출을 둘 다 가지고 있는 모든 고객의 이름을 구하라

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

- 계좌는 가지고 있지만 대출을 가지고 있지 않은 모든 고객의 이름을 구하라

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

5. 집계 함수(aggregate functions)

Ch 3. SQL

집계 함수 (1)

- 이 함수는 입력으로 값들의 집합(집합 또는 다중 집합)을 가지며, 결과값으로는 단일 값을 갖는 함수

  avg: 평균

  min: 최소값

  max: 최대값

  sum: 총합

  count: 개수

집계 함수 (2)

- Perryridge 지점의 평균 잔고를 구하라

```
select avg(balance)
from account
where branch_name = 'Perryridge'
```

- customer 릴레이션에서 투플의 개수를 구하라

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

- 은행에서 예금자의 수를 구하라

```
select count(distinct customer_name)
from depositor
```

Group By

- 각 지점의 예금자들의 수를 구하라

```
select branch_name, count (distinct customer_name)
from depositor, account
where depositor.account_number=account.account_number
group by branch_name
```

- Note: **select** 절에서 집계 함수 밖에 있는 속성은 반드시 **group by** 목록에 나타나야 함

Having 절

- 평균 잔고가 $1,200 이상인 지점들을 찾아라

```
select branch_name, avg (balance)
from account
group by branch_name
having avg (balance) > 1200
```

  - **where** 절과 **having** 절이 같은 질의어에 나타나게 되면, SQL은 **where** 절의 조건을 먼저 적용
  - **where** 조건을 만족하는 투플들은 그 후 group by절에 의해 그룹으로 묶이게 됨
  - SQL은 있다면 having 절을 그 후에 각 그룹에 대해 적용하며, 여기에서 having 절의 조건을 만족하지 않는 그룹을 제거
  - select 절은 질의의 결과 투플들을 생성하기 위해 남은 그룹들을 사용

6. 널 값

Ch 3. SQL

널 값

- null이라는 키워드를 통해 널 값을 가진 투플을 나타낼 수 있음
- 널은 알지 못하는(unknown) 값이거나, 존재하지 않는(not exist) 값을 나타냄
- 술어 **is null**은 널값을 체크하는데 사용
  - Ex) loan 릴레이션에서 amount가 널값인 모든 loan number를 구하라

```
select loan_number
from loan
where amount is null
```

- 널을 포함한 수학적 표현식의 결과는 널
  - E.g. 5 + null의 결과는 널
- 그러나, 집계 함수는 간단하게 널을 무시

널 값과 부울 연산(Boolean operation)

- 널에 대한 비교는 unknown
  - Ex) 5 < null 또는 null <> null 또는 null = null

- 부울 연산의 정의가 unknown값을 다룰 수 있도록 확장됨
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*, (*false* **and** *unknown*) = *false,*
    (*unknown* **and** *unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*
  - "*P* **is unknown"**은 술어 P가 unknown 이라면 true라는 의미

- **where**절 술어의 결과가 unknown 이라면 false로 간주

널 값과 집계 함수

- 모든 대출 총액의 합계를 구하라

```
select sum (amount)
from loan
```

  - 위의 문장에서 amount 값이 널인 경우 무시
  - 널이 아닌 값이 없다면 결과는 널

- **count(*)**를 제외한 모든 집계 연산은 널 값을 무시

7. 중첩 하위 질의(nested subqueries)

Ch 3. SQL

중첩 하위 질의

- SQL은 중첩된 하위 질의를 허락하기 위한 메커니즘을 제공함
- 하위 질의는 다른 질의 안에 중첩된 **select-from-where** 표현
- 하위 질의의 일반적인 용도는 집합의 멤버쉽을 테스트하고, 집합 비교를 하고, 집합의 대응수(cardinality)를 결정하는 데 사용

질의어 예제 (1)

- 은행에서 계좌와 대출 두 가지를 가진 모든 고객을 구하라

select distinct *customer_name*
from *borrower*
where *customer_name* **in**
(**select** *customer_name* **from** depositor)

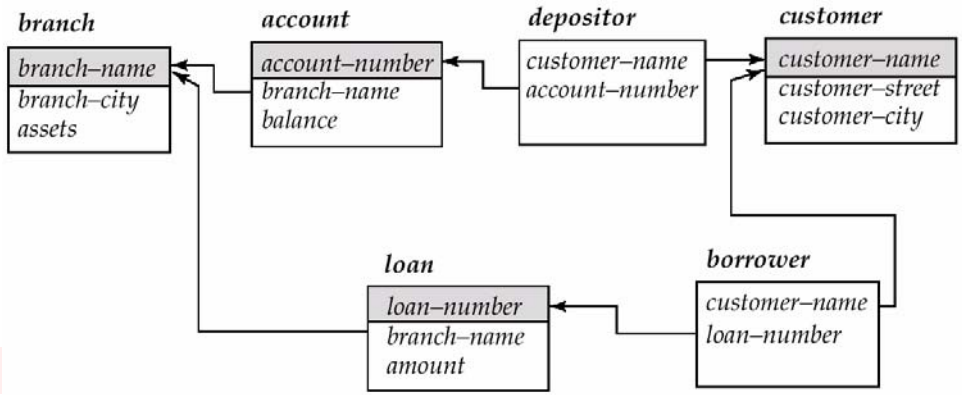
- 은행에 대출을 가지고 있지만 계좌는 가지고 있지 않은 모든 고객을 구하라

select distinct *customer_name*
from *borrower*
where *customer_name* **not in**
(**select** *customer_name* **from** *depositor*)

질의어 예제 (2)

- Perryridge 지점에 계좌와 대출을 동시에 가지고 있는 고객을 구하라

select distinct *customer-name*
from *borrower, loan*
where *borrower.loan_number = loan.loan_number* **and**
branch_name = 'Perryridge' **and**
(*branch_name, customer_name*) **in**
(select *branch_name, customer_name*
from *depositor, account*
where *depositor.account_number = account.account_number*)



집합 비교

- Brooklyn에 위치하는 지점 중 최소한 하나 이상의 지점보다 자산이 큰 모든 지점의 이름을 구하라

```
select distinct  T.branch_name
from branch as T, branch as S
where  T.assets > S.assets and
                  S.branch_city = 'Brooklyn'
```

- "하나 이상보다 큰"이라는 의미로 **> some** 절이 사용될 수 있음. 이 질의어는 위와 같은 의미의 질의어

```
select branch_name
from branch
where assets > some
        (select assets
         from branch
         where branch_city = 'Brooklyn')
```

Some 절의 정의

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ s.t. (F <comp> t)
  Where <comp> can be: $<, \leq, >, =, \neq$

(5< **some** | 0 / 5 / 6 |) = true
(read: 5 < some tuple in the relation)

(5< **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≡ **not in**

All 절의 정의

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> t)

(5< **all** {0, 5, 6}) = false

(5< **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≢ **in**

질의어 (3)

- Brooklyn에 있는 각 지점보다 큰 자산을 갖는 모든 지점들의 이름을 찾아라

```
select branch_name
from branch
where assets > all
        (select assets
        from branch
        where branch_city = 'Brooklyn')
```

빈 릴레이션에 대한 테스트

- **exists** 구문은 인자의 하위 질의가 비어있지 않을 때 **참**를 반환
- **exists** $r \Leftrightarrow r \neq \varnothing$
- **not exists** $r \Leftrightarrow r = \varnothing$

질의어 (4)

- Brooklyn에 위치하는 모든 지점에 계좌를 가진 모든 고객을 구하라

```
select distinct S.customer_name
from depositor as S
where not exists (
        (select branch_name
        from branch
        where branch_city = 'Brooklyn')
        except
        (select R.branch_name
        from depositor as T, account as R
        where T.account_number = R.account_number and
              S.customer_name = T.customer_name))
```

- X - Y = Ø 은 X ⊆ Y 임을 나타냄

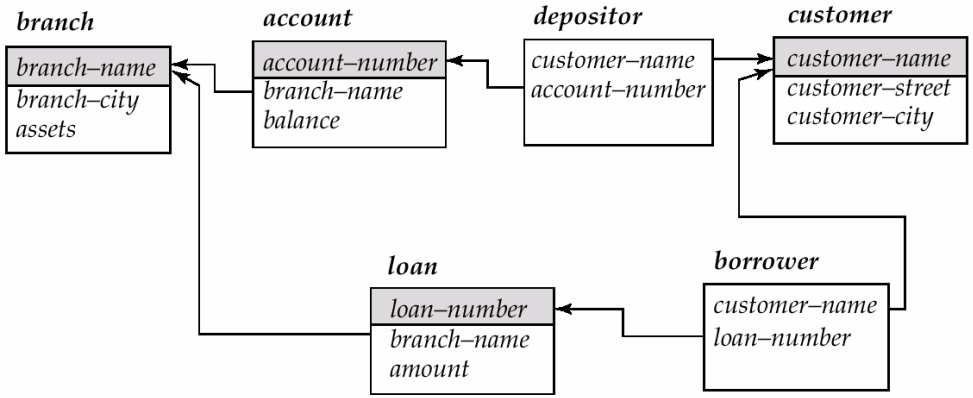
- Note: 이 질의어는 = **all**을 쓸 수 없음

중복 투플 부재에 대한 테스트

- **unique** 구문은 만일 인자로 주어진 하위 질의가 중복된 투플을 가지지 않는다면 투플이 됨

- Perryridge 지점에서 최대 하나의 계좌를 가진 모든 고객들을 구하라

```
select T.customer_name
from depositor as T
where unique (
    select R.customer_name
    from account, depositor as R
    where T.customer_name = R.customer_name and
       R.account_number = account.account_number and
       account.branch_name = 'Perryridge')
```



질의어 예제 (5)

- Perryridge 지점에서 적어도 두 개 이상의 계좌를 가진 모든 고객을 구하라

```
select distinct T.customer_name
from depositor T
where not unique (
    select R.customer_name
    from  account, depositor as R
    where T.customer_name = R.customer_name and
          R.account_number = account.account_number and
          account.branch_name = 'Perryridge')
```

8. 복합 질의(Complex Query)

Ch 3. SQL

유도 릴레이션

- 계좌 잔고 평균이 $1200보다 큰 지점들의 평균 계좌 잔고를 구하라

```
select branch_name, avg_balance
from (select branch_name, avg (balance)
      from account
      group by branch_name)
      as result (branch_name, avg_balance)
where avg_balance > 1200
```

from 절의 결과가 임시 뷰 릴레이션으로 되기 때문에, **having** 절을 사용할 필요가 없음. 따라서, **where** 절에서 결과의 속성을 바로 사용할 수 있음

With 절

- with 절은 그것이 나타난 질의어에서만 유효한 임시 뷰를 정의할 수 있게 함.이는 프로그램이 수행하는 하나의 작업을 여러 개의 프로시저로 나누어 구성하는 것과 같음

- 최대 잔고를 가진 모든 계좌를 구하라

```
with max_balance(value) as
     select max(balance)
     from account
select account_number
from account, max_balance
where account.balance = max_balance.value
```

With 절을 사용한 복잡한 쿼리

- 모든 지점의 총 계좌 예금의 평균보다 많은 총 계좌 예금을 갖는 모든 지점을 구하라

```
with branch_total (branch_name, value) as
   select branch_name, sum (balance)
   from account
   group by branch_name
with branch_total_avg(value) as
   select avg (value)
   from branch_total
select branch_name
from branch_total, branch_total_avg
where branch_total.value >= branch_total_avg.value
```

9. 뷰(view)

Ch 3. SQL

뷰

- 특정 사용자의 관점으로부터 특정 데이터를 숨기기 위한 메카니즘. 뷰를 만들기 위해서는 다음과 같은 형식을 사용

  create view *v* **as** <query expression>

  - <query expression>은 적법한 질의 표현
  - 뷰의 이름은 *v*로 표현

질의어 예제

- 지점과 그들의 고객을 구성하는 뷰는 다음과 같음

```
create view all_customer as
 (select branch_name, customer_name
  from depositor d, account a
  where d.account_number = a.account_number)
  union
 (select branch_name, customer_name
  from borrower, loan
  where borrower.loan-number = loan.loan_number)
```

- Perryridge 지점에서 모든 고객을 구하라

```
select customer_name
from all_customer
where branch_name = 'Perryridge'
```

다른 뷰를 이용한 뷰 정의

- 다른 뷰를 정의하는 식에서도 뷰 사용

```
create view perryridge_customer as
select customer_name
from all_customer
where branch_name = 'Perryridge'
```

- 뷰 확장(view extension)
  - v_1, v_2, v_3가 순환적으로 정의

```
repeat
   Find any view relation v_i in e_1
   Replace the view relation v_i by the expression defining v_i
until no more view relation are present in e_1
```

10. 데이터베이스의 변경

Ch 3. SQL

삭제

- Perryridge 지점의 모든 계좌 정보를 삭제하라

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

- Needham 시에 위치한 모든 지점의 모든 계좌를 삭제하라

```
delete from account
where branch_name in
    (select branch_name
     from branch
     where branch_city = 'Needham')
delete from depositor
where account_number in
    (select account_number
     from branch, account
     where branch_city = 'Needham'
       and branch.branch-name = account.branch_name)
```

질의어 예제

- 은행에서 평균 이하의 계좌 잔고를 가진 모든 계좌를 삭제하라

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

  - 문제점: deposit에서 투플을 삭제한다면, 평균 잔고가 변경되게 됨
  - SQL에서의 문제점 해결:
    1. 먼저, 평균 잔고를 계산하고, 삭제할 모든 투플을 찾음
    2. 다음으로는, 위에서 찾은 모든 투플들을 삭제 (평균에 대한 재계산이나 투플에 대한 재 테스트없이 함)

삽입

- 계좌에 새로운 투플을 추가

```
insert into account
    values ('A-9732', 'Perryridge',1200)
```

또는 동등하게 다음과 같이 표현

```
insert into account(branch_name,balance,account_number)
    values ('Perryridge', 1200, 'A-9732')
```

- 잔고가 널인 새로운 투플을 계좌에 추가

```
insert into account
    values ('A-777','Perryridge', null)
```

삽입

- Perryridge 지점의 각각의 대출에 대해 $200의 예금을 가진 새로운 계좌를 대출 고객들에게 선물로 지급하기를 원한다고 한다. 이 때 대출번호가 예금 계좌의 계좌번호가 된다고 하자

```
insert into account
   select loan_number, branch_name,  200
   from loan
   where branch_name = 'Perryridge'
insert into depositor
   select customer_name, loan_number
   from loan, borrower
   where branch_name = 'Perryridge'
     and loan.account_number=borrower.account_number
```

- 릴레이션에 삽입을 하기 전에 select문을 모두 수행해야 함(그렇지 않으면 질의어는 문제를 발생)

```
insert into table1 select * from table1
```

갱신

- $10,000 이상의 계좌는 6%, 다른 모든 계좌는 5%의 이자를 받는다고 하자
  - 다음과 같은 두 개의 **갱신** 질의어를 작성할 수 있음

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

```
update account
set balance = balance * 1.05
where balance ≤ 10000
```

  - update문은 순서가 중요
  - update문의 순서와 관련된 문제를 피할 수 있도록 **case** 구문을 사용할 수 있음

조건에 대한 갱신을 하는 Case 문

- 이는 앞장과 같은 의미의 질의어. $10,000 이상의 계좌는 6%, 다른 모든 계좌는 5%의 이자를 받는다고 하자

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

뷰의 갱신

- amount 속성을 숨기고, loan 릴레이션의 모든 loan 데이터를 나타내는 뷰를 생성한 것은 다음과 같음

```
create view branch-loan as
        select branch-name, loan-number
        from loan
```

- branch-loan에 다음과 같이 새로운 투플을 추가

```
insert into branch-loan
        values ('Perryridge', 'L-307')
```

  - 이 삽입은 반드시 loan 릴레이션에 투플('L-307', 'Perryridge', null)을 삽입함으로서 표현해야 함

- 더 복잡한 뷰의 갱신를 할 경우, 다루기가 더 어렵거나 복잡하기 때문에, 제약조건을 강제하고 있음

- 대부분의 SQL 구현은 논리 데이터베이스에서 통합화(aggregation)를 사용하지 않고, 하나의 릴레이션으로 정의된 간단한 뷰에서만 허용함

트랜잭션(Transactions) (1)

- 트랜잭션은 일련의 질의와 갱신문으로 구성
  - SQL문이 실행되는 순간에 암묵적으로 트랜잭션이 시작되고, 다음 중의 하나로 끝나야 함
    - **commit work:** 트랜잭션에 의해 수행된 갱신들을 데이터베이스에 영구적으로 반영
    - **rollback work:** 트랜잭션에 의해 수행된 모든 갱신들을 취소(undo)시킴

- 예제
  - 현금을 한 계좌에서 다른 계좌로 이체시킬 경우 다음의 두 작업이 있음
    - 한 계좌에서 금액을 공제한 후, 다른 계좌로 예금
  - 위의 절차 중 한 가지만 성공하고 다른 한가지는 실패한다면, 데이터베이스는 모순된 상태에 있음
  - 그러므로, 두 개 모두 성공하거나, 두 개 모두 실행되지 않아야 함

- 트랜잭션의 어떤 절차라도 실패한다면, 트랜잭션에 의해 수행된 모든 작업은 **rollback work**에 의해 취소할 수 있음

- 시스템 실패(failure)와 같은 경우, 불완전한 트랜잭션의 rollback은 자동적으로 수행

트랜잭션 (2)

- 대부분의 데이터베이스 시스템에서, 성공적으로 수행된 각각의 SQL 문은 자동적으로 commit
  - 각각의 트랜잭션은 기본적으로 하나의 SQL문의 구성으로 됨
  - 각 SQL문의 자동 완료(automatic commit)는 트랜잭션이 여러 개의 SQL 문으로 구성되는 경우에는 해제되어야 함. 그러나 어떻게 자동 완료를 해제할 것인가는 데이터베이스 시스템에 종속적
  - SQL:1999의 다른 대안: 다음의 키워드로 문장을 감싸도록 하여 지원

```
begin atomic
 ...
end
```

11. 조인된 릴레이션

예제

- 내부 조인
  - *loan* **inner join** *borrower* **on** *loan.loan_number = borrower.loan_number*
  - *loan* **inner join** *borrower* **on** *loan.loan_number = borrower.loan_number*

    as *lb*(*loan_number*, *branch*, *amount*, *cust*, *cust_loan_num*)

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

예제

- Left outer join
  - *loan* **left outer join** *borrower* **on** *loan.loan_number = borrower.loan_number*

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

- 자연조인
  - *loan* **natural inner join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

조인 타입과 조건 (1)

- Join types
  - **inner join**
  - **left outer join**
  - **right outer join**
  - **full outer join**
- Join conditions
  - **natural**
    - 조인 속성의 순서 (조인 속성, 좌측항 비조인 속성, 우측항 비조인 속성)
  - **on** <predicate>
  - **using** (A_1, A_2, …, A_n)
    - 조인 속성이 두 릴레이션의 공통된 모든 속성들이 아니라 A_1 ~ A_n
    - A_1, A_2, …, A_n : 두 릴레이션의 공통된 속성으로만
    - 조인의 결과에 한번만 나타나야

조인 타입과 조건 (2)

- 예제
  - *loan* **natural right outer** *join borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

  - *loan* **full outer join** *borrower* **using** (*loan_number*)

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

조인 타입과 조건 (3)

- 은행에 계좌를 가지고 있지만 대출은 없는 모든 고객들을 구하라
  - **select** *d_CN*

    from (*depositor* **left outer join** *borrower*

    on *depositor.customer_name* = *borrower.customer_name*)

    as *db1* (*d_CN*, *account_number*, *b_CN*, *loan_number*)

    where *b_CN* **is null**
- 은행에 계좌나 대출 둘 중 하나만을 가지는 고객들을 모두 구하라
  - **select** *customer_name*

    from (*depositor* **natural full outer join** *borrower*)

    where *account_number* **is null or** *loan_number* **is null**

Ch 3. SQL

1. 배경
2. 기본 구조
3. 집합 연산
4. 집계 함수(aggregate functions)
5. 널 값
6. 중첩 하위 질의(nested subqueries)
7. 뷰(view)
8. 복합 질의(complex query)
9. 데이터베이스의 변경
10. 조인된 릴레이션