2021. 10. 11

dongguk UNIVERSITY

5. 고급 SQL-02

Chapter 5: Advanced SQL-02

동국대학교 과학기술대학
컴퓨터공과 DB&WE lab
변정용
byunjy@dongguk.ac.kr

목차 Contents

- 프로그래밍 언어(고급언어) SQL 접근
- **함수와 프로시저**
- 격발기 Triggers
- 재귀적 질의
- 고급 집계 기능 Advanced Aggregation Features
- OLAP

함수와 프로시저

- SQL:1999는 함수와 프로시저를 지원한다.
  - ✓ 함수/프로시저는 SQL 또는 외부 프로그래밍언어로 쓰여질 수 있다.
  - ✓ 함수는 특히 영상, 기하적 대상과 같은 특수화된 자료형에 유용하다.
    - 예: 다각형이 겹치는 지 여부 점검 또는 이미지 유사성 비교하는 함수
  - ✓ 몇몇 DB 시스템은 결과로써 관계를 반환하는 테이블-값 함수를 지원한다.
- SQL:1999 는 또한 반복, if-then-else, 할당문을 포함한 풍부한 명령 구조 집합을 지원한다.
- 많은 데이터베이스는 SQL-1999와 다른 SQL로 적정한 절차적 확장을 가지고 있다.

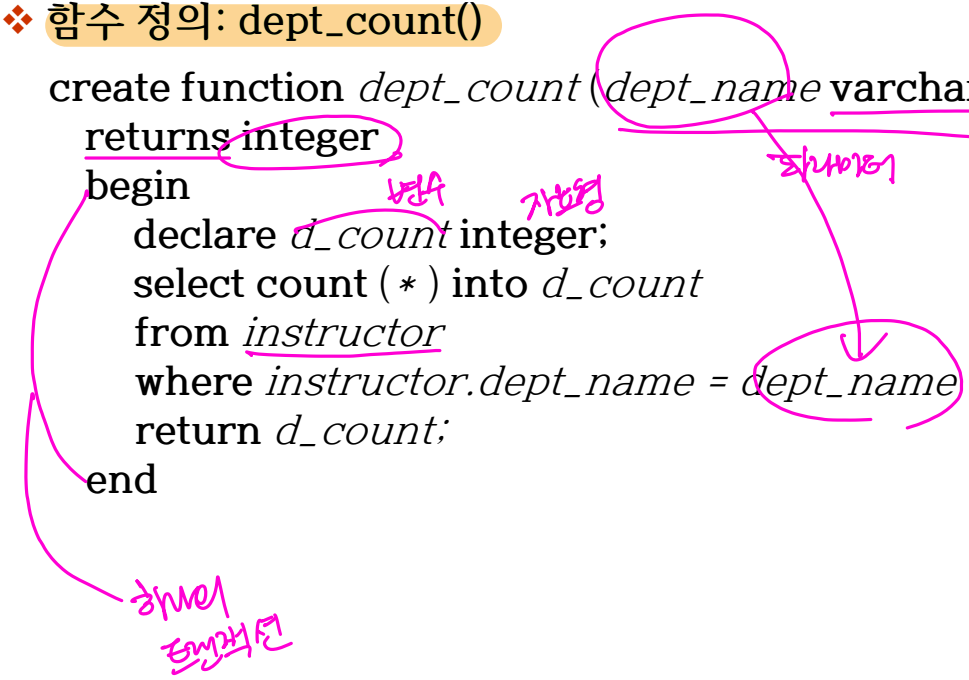
함수와 프로시저

- 데이터베이스 내부에 SQL 함수와 프로시저를 저장하고, 호출하는 방법 제공
- 프로시저와 함수 관련 대학 DB 비즈니스 논리 예
  - ✓ 한 학생이 주어진 학기에 수강한 과목 수 검색
  - ✓ 전임 교수의 연간 강의 수
  - ✓ 한 학생이 등록 가능한 최대 (복수) 전공의 수
- SQL은 함수, 프로시저, 메소드의 정의를 허용한다.
- C, C++, Java 언어에 의하여 정의된다.
- 비표준 문법 구현
  - ✓ Oracle(PL/SQL), MS SQL Server(TransactSQL), PostgreSQL(PL/pgSQL)

함수와 프로시저 : 선언과 호출 1

❖ 함수 정의: dept_count()

```
create function dept_count (dept_name varchar(20))
    returns integer
    begin
        declare d_count integer;
        select count (*) into d_count
        from instructor
        where instructor.dept_name = dept_name
        return d_count;
    end
```



함수와 프로시저:선언과 호출 2

- 12명 보다 많은 교수를 가진 학과이름과 예산을 찾으시오.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12;
```

- 함수 이름의 접두사 instructor_of.dept_name으로 해서 참조

```
select *
from table(instructor_of('Finance'));
```

- 함수 값을 가지는 테이블을 사용 않고 질의 직접 작성
- 일반적으로 함수 값을 갖는 테이블은 매개변수 허용, 매개변수를 가진 뷰(parameterized view)로 간주된다.

함수와 프로시저: Table 함수

- SQL:2003는 관계를 반환하는 함수를 추가했다.
- 예제: 주어진 고객에 의해 소유된 모든 계정을 반환하라.

```
create function instructors_of (dept_name char(20)
    returns table ( ID varchar(5),
                    name varchar(20),
                      dept_name varchar(20),
                    salary numeric(8,2))
    return table
            (select ID, name, dept_name, salary
             from instructor
            where instructor.dept_name = instructors_of.dept_name)
```

- 용법

```
select *
from table (instructors_of ('Music'))
```

격발 Triggers

격발 Triggers

- 격발 **trigger** 은 DB 수정의 측면효과(side effect)로서 시스템에 의하여 자동으로 수행되는 문장이다.
- 격발 장치를 설계하기 위하여, 우린 다음을 서술해야 한다: ECA(Event-Condition-Action)
  - ✓ 격발 시점 명시 곧, 검사되어야 할 사건(event)과 격발이 만족되어야 할 조건(condition).
  - ✓ 격발이 수행될 때 취해야 할 행동(action)
- 사용자 책임 :
  - ✓ 격발을 DB에 입력 > 명시된 사건 발생 > 조건 만족 > 행동
- 격발 Triggers 은 SQL:1999에서 표준으로 소개되었지만, 대부분의 DB에 의해서 비표준 구문으로 좀 더 일찍이 지원되었다.
  - ✓ 여기서 그려진 구문은 여러분의 DB 시스템에 정확하게 작동하지 않을 수도 있다. 시스템 안내서를 점검하시오.

격발

- 트리거의 필요성
  - ✓ 트리거는 SQL 제약조건을 사용할 수 없는 무결성 제약조건 구현용도
  - ✓ 알림의 자동화 수단으로 유용
  - ✓ 예,
    - takes.grade에 값이 할당될 때마다 student.tot_cred 자동계산 자동화
    - 재고물량 최저 값에 이르면 주문 요구하는 투플 추가 자동화

격발기(Trigger) 예제

- 예. *time_slot_id* 는 *timeslot*의 기본키가 아니다. 그래서 *section* 에서 *timeslot*으로 외래 키 제약조건을 생성할 수 없다.
- 편법: 무결성 제약조건을 이행하기 위하여 *section* 과 *timeslot* 에 관한 격발기들을 사용한다.

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
        select time_slot_id
        from time_slot)) /* time_slot에 나타나진 않은 time_slot_id */
begin
    rollback
end;
```

격발기 예제 계속

```
create trigger timeslot_check2 after delete on timeslot
    referencing old row as orow
    for each row
    when (orow.time_slot_id not in (
                    select time_slot_id
                    from time_slot)
                    /* last tuple for time slot id deleted from time slot */
                and orow.time_slot_id in (
                    select time_slot_id
                    from section)) /* and time_slot_id still referenced from section*/
    begin
        rollback
    end;
```

SQL에서 사건과 행동을 격발하기

- 사건을 격발하기는 **insert, delete 또는 update 될 수 있다.**
- update에 관한 격발기들은 특정 속성들로 제한될 수 있다.
  - ✓ **예,** *takes*의 *grade*에 관한 갱신 이후에
- 갱신이 참조될 수 있기 전후에 속성들의 값들
  - ✓ **referencing old row as** : 삭제와 갱신을 위하여
  - ✓ **referencing new row as** : 삽입과 갱신을 위하여
- 어떤 사건 전에 격발기들은 작동될 수 있다. 그것은 보조 제약조건들로써 서비스될 수 있다. 예. 공백 학점을 null로 번역한다.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
 begin atomic
        set nrow.grade = null;
 end;
```

credits_earned 값을 유지하기 위한 격발기

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred= tot_cred +
        (select credits
         from course
         where course.course_id= nrow.course_id)
    where student.id = nrow.id;
end;
```

격발기가 부적합한 경우

- 격발기들이 작업들에 대하여 더 일찍 사용되었던 사항들:
  - ✓ 요약 자료 유지하기(예, 각 학과의 전체봉급)
  - ✓ DB 사본 만들 때 트리거 이용 - 중복하기(**change** 또는 **delta**)로 변화 기록.
  - ✓ DB 사본 만들 때 맞춤형 기능 지원 – 불 필요
- 실체화 뷰 유지 용도:
  - ✓ 각 수업에 등록한 전체 학생 수에 빠른 접근 지원.
    - Section_registration(course_id,sec_id,semester,year,total_students)

      select course_id,sec_id,semester,year,count(ID) as **total_students**

      from takes group by course_id,sec_id,semester,year;
    - **total_students** 값: takes에 삽입,삭제,갱신에 대한 트리거로 유지되어야 한다.
  - ✓ 현대 대부분 DBMS는 실체화 뷰를 지원하므로 트리거 필요치 않다.

격발기가 부적합한 경우

- 격발기들의 의도하지 않은 수행의 위험, 경우 예:
  - ✓ 백업 사본으로부터 자료를 적재하기(싣기)
  - ✓ 원격 지에서 갱신들을 중복하기
  - ✓ 격발기 수행은 그런 행동 전에 무력화될 수 있다.
- 격발기가 가진 다른 위험들:
  - ✓ 격발기를 해제하는 중요한 트랜잭션들의 실패로 이끄는 오류
  - ✓ 연쇄적인 수행

고급 집계 기능
Advanced Aggregation Features

P176

순위화 Ranking

- 순위화는 기본 SQL 집계를 사용해서 효율적 구현이 어려운 작업들이 있다.
  - ✓ student_grades(ID,GPA) 뷰 활용 학점 순위화

```
select ID, rank() over (order by (GPA) desc) as s_rank
from student_grades;
order by s_rank;
```

  - ✓ 기본 SQL 집계함수

```
select ID, (1 + (select count(*)
                 from student_grades B
                 where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

Ranking (Cont.)

- 각 구역별로 학생들의 순위를 제공한다.
- "각 학과내에 학생의 순위를 찾아라."

```
select ID, dept_name,
    rank () over (partition by dept_name order by GPA desc)
            as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

- Multiple **rank** clauses can occur in a single **select** clause.
- Ranking is done *after* applying **group by** clause/aggregation
- Can be used to find top-n results
  - ✓ More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

Ranking (Cont.)

Other ranking functions:

- **percent_rank** (within partition, if partitioning is done)
- **cume_dist** (cumulative distribution)
  - fraction of tuples with preceding values
- **row_number** (non-deterministic in presence of duplicates)

SQL:1999 permits the user to specify **nulls first** or **nulls last**

select *ID*,
    rank ( ) **over** (**order by** *GPA* **desc nulls last**) **as** *s_rank*
from *student_grades*

Ranking (Cont.)

For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.

E.g.,

select *ID*, **ntile**(4) **over** (**order by** *GPA* **desc**) **as** *quartile*
from *student_grades;*

Windowing

Used to smooth out random variations.

E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”

Window specification in SQL:

Given relation *sales(date, value)*

select *date,* ***sum***(*value*) **over**
(**order by** *date* **between rows** 1 **preceding and** 1 **following**)
from *sales*

Windowing

Examples of other window specifications:

between rows unbounded preceding and current

rows unbounded preceding

range between 10 **preceding and current row**

- All rows with values between current row value –10 to current value

range interval 10 **day preceding**

- Not including current row

Windowing (Cont.)

Can do windowing within partitions

E.g., Given a relation *transaction* (*account_number, date_time, value*), where value is positive for a deposit and negative for a withdrawal

"Find total balance of each account after each transaction on the account"

```
select account_number, date_time,
    sum (value) over
            (partition by account_number
            order by date_time
            rows unbounded preceding)
      as balance
from transaction
order by account_number, date_time
```

온라인 분석 처리
OLAP**

p180

5.25

Data Analysis and OLAP

- **온라인 분석처리 (OLAP)**
  - ✓ 다차원 데이터의 서로 다른 요약을 볼 수 있도록 해주는 대화형 시스템
  - ✓ 요약 정보 요청이 수초 내에 곧 바로 얻어져야 함
- 자료는 다차원 자료(**multidimensional data)**라는 차원 속성과 척도로써 모델화 될 수 있다.
  - 몇몇 값을 척도화 한다.
  - 집계될 수 있다.
    - 예, *sales* 관계의 속성 수
  - ✓ **차원 속성 Dimension attributes**
    - 척도 속성(집계)이 보여질 차원을 정의한다.
      - 예, sales 관계의 속성 *item_name, color, size*

Example sales relation

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6
...	...	...	...
...	...	...	...

item_name과 color에 의한 sales cross-tab

clothes_size **all**

item_name	*color* dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

The table above is an example of a **cross-tabulation** (**cross-tab**), also referred to as a **pivot-table**.

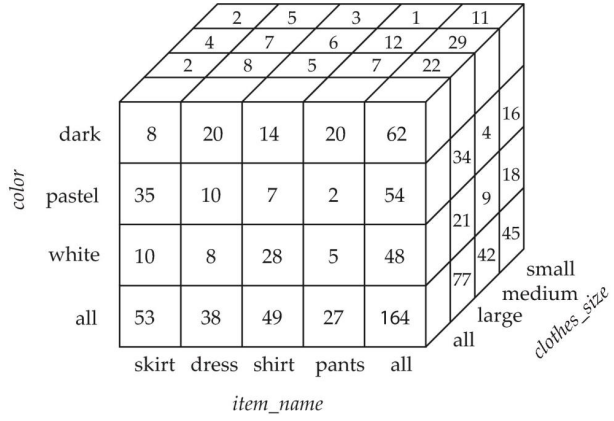
- Values for one of the dimension attributes form the row headers
- Values for another dimension attribute form the column headers
- Other dimension attributes are listed on top
- Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

Data Cube

A **data cube** is a multidimensional generalization of a cross-tab

Can have *n* dimensions; we show 3 below

Cross-tabs can be used as views on a data cube



Cross Tabulation With Hierarchy

Cross-tabs can be easily extended to deal with hierarchies

- Can drill down or roll up on a hierarchy

clothes_size: **all**

category	*item_name*	*color*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Relational Representation of Cross-tabs

Cross-tabs can be represented as relations

- We use the value **all** is used to represent aggregates.
- The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	**all**	8
skirt	pastel	**all**	35
skirt	white	**all**	10
skirt	**all**	**all**	53
dress	dark	**all**	20
dress	pastel	**all**	10
dress	white	**all**	5
dress	**all**	**all**	35
shirt	dark	**all**	14
shirt	pastel	**all**	7
shirt	White	**all**	28
shirt	**all**	**all**	49
pant	dark	**all**	20
pant	pastel	**all**	2
pant	white	**all**	5
pant	**all**	**all**	27
all	dark	**all**	62
all	pastel	**all**	54
all	white	**all**	48
all	**all**	**all**	164

Extended Aggregation to Support OLAP

The **cube** operation computes union of **group by**'s on every subset of the specified attributes

Example relation for this section
sales(*item_name, color, clothes_size, quantity*)

E.g. consider the query

select *item_name, color, size,* **sum**(*number*)
from *sales*
group by cube(*item_name, color, size*)

This computes the union of eight different groupings of the *sales* relation:

{ (*item_name, color, size*), (*item_name, color*),
(*item_name, size*), (*color, size*),
(*item_name*), (*color*),
(*size*), ( ) }

where ( ) denotes an empty **group by** list.

For each grouping, the result contains the null value
for attributes not present in the grouping.

Online Analytical Processing Operations

Relational representation of cross-tab that we saw earlier, but with *null* in place of **all**, can be computed by

select *item_name*, *color*, **sum**(*number*)
from *sales*
group by cube(*item_name, color*)

The function **grouping()** can be applied on an attribute

Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

select *item_name, color, size*, **sum**(*number*),
grouping(*item_name*) **as** *item_name_flag*,
grouping(*color*) **as** *color_flag*,
grouping(*size*) **as** *size_flag*,
from *sales*
group by cube(*item_name, color, size*)

Online Analytical Processing Operations

Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**

E.g., replace *item_name* in first query by

decode(**grouping**(item_*name*), 1, ‘all’, *item_name*)

Extended Aggregation (Cont.)

The **rollup** construct generates union on every prefix of specified list of attributes

E.g.,

select *item_name*, *color*, *size*, **sum**(*number*)
from *sales*
group by rollup(*item_name, color, size*)

Generates union of four groupings:

{ (*item_name, color, size*), (*item_name, color*), (*item_name*), ( ) }

Rollup can be used to generate aggregates at multiple levels of a hierarchy.

E.g., suppose table *itemcategory*(*item_name, category*) gives the category of each item. Then

select *category, item_name*, **sum**(*number*)
from *sales, itemcategory*
where *sales.item_name = itemcategory.item_name*
group by rollup(*category, item_name*)

would give a hierarchical summary by *item_name* and by *category.*

Extended Aggregation (Cont.)

Multiple rollups and cubes can be used in a single group by clause

Each generates set of group by lists, cross product of sets gives overall set of group by lists

E.g.,

select *item_name, color, size*, **sum**(*number*)
from *sales*
group by rollup(*item_name*), **rollup**(*color, size*)

generates the groupings

{*item_name, ()*} *X* {*(color, size), (color), ()*}

= { (*item_name, color, size*), (*item_name, color*), (*item_name*), (*color, size*), (*color*), ( ) }

Online Analytical Processing Operations

Pivoting: changing the dimensions used in a cross-tab is called

Slicing: creating a cross-tab for fixed values only

Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.

Rollup: moving from finer-granularity data to a coarser granularity

Drill down: The opposite operation - that of moving from coarser-granularity data to finer-granularity data

OLAP Implementation

The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems.

OLAP implementations using only relational database features are called **relational OLAP (ROLAP)** systems

Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.

OLAP Implementation (Cont.)

- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
  - ✓ Space and time requirements for doing so can be very high
    - 2^n combinations of **group by**
  - ✓ It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
      - For all but a few "non-decomposable" aggregates such as *median*
      - is cheaper than computing it from scratch
- Several optimizations available for computing multiple aggregates
  - ✓ Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
  - ✓ Can compute aggregates on (*item_name, color, size*), (*item_name, color*) and (*item_name*) using a single sorting of the base data

재귀적 질의
Recursive Queries

P171 교재

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - ✓ Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

Computing transitive closure using iteration, adding successive tuples to *rec_prereq*

- The next slide shows a *prereq* relation
- Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
- The final result is called the *fixed point* of the recursive view definition.

Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

End of Chapter

Figure 5.22

item_name	*clothes_size*	*dark*	*pastel*	*white*
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
pant	small	14	1	3
pant	medium	6	0	0
pant	large	0	1	2

Figure 5.23

item_name	*quantity*
skirt	53
dress	35
shirt	49
pant	27

Figure 5.24

item_name	*color*	*quantity*
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pant	dark	20
pant	pastel	2
pant	white	5

또 다른 재귀 예제

- ✓ 주어진 relation
  manager(*employee_name, manager_name*)
- ✓ 모든 직업 경영자 짝을 찾아라. 여기서 고용인은 직.간접으로 경영자에게 보고한다.(즉, 경영자의 경영자, 경영자의 경영자의 경영자 등)

```
with recursive empl (employee_name, manager_name ) as (
        select employee_name, manager_name
        from   manager
    union
        select manager.employee_name, empl.manager_name
        from   manager, empl
        where manager.manager_name = empl.employe_name)
select *
from   empl
```

이 예제 뷰, *empl,* 은 i*manager* relation의 이행적 폐포(*transitive closure)*이다.

통합(Merge) 문장 (제24장에서)

- Merge 구조는 갱신의 순차 처리를 허용한다.
- 예제: relation *funds_received* (*account_number, amount*)는 has batch of deposits to be added to the proper account in the *account* relation

```
merge into account as A
    using (select *
           from funds_received as F )
      on (A.account_number = F.account_number )
      when matched then
          update set balance = balance + F.amount
```