



Lecture 6 of 42

Modern SQL: Cursors and Views

Wednesday, 06 September 2006

William H. Hsu

Department of Computing and Information Sciences, KSU

KSOL course page: <http://snipurl.com/va60>

Course web site: <http://www.kddresearch.org/Courses/Fall-2006/CIS560>

Instructor home page: <http://www.cis.ksu.edu/~bhsu>

Reading for Next Class:

Sections 4.1 – 4.2, p. 121 – 132, Silberschatz *et al.*, 5th edition

Sections 4.3 – 4.5, p. 133 – 145, Silberschatz *et al.*, 5th edition



Bank Example Queries: Review

- Find all customers who have an account from at least the “Downtown” and the Uptown” branches.

- Query 1

$$\Pi_{customer_name} (\sigma_{branch_name = \text{“Downtown”}} (depositor \bowtie account)) \cap$$

$$\Pi_{customer_name} (\sigma_{branch_name = \text{“Uptown”}} (depositor \bowtie account))$$

- Query 2

$$\Pi_{customer_name, branch_name} (depositor \bowtie account)$$

$$\div \rho_{temp(branch_name)} (\{ \text{“Downtown”}, \text{“Uptown”} \})$$

Note that Query 2 uses a constant relation.





Deletion: Review

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.



Deletion Examples: Review

- Delete all account records in the Perryridge branch.
 $account \leftarrow account - \sigma_{branch_name = "Perryridge"}(account)$

- Delete all loan records with amount in the range of 0 to 50
 $loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}(loan)$

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{branch_city = "Needham"}(account \bowtie branch)$$

$$r_2 \leftarrow \Pi_{branch_name, account_number, balance}(r_1)$$

$$r_3 \leftarrow \Pi_{customer_name, account_number}(r_2 \bowtie depositor)$$

$$account \leftarrow account - r_2$$

$$depositor \leftarrow depositor - r_3$$





Insertion: Review

- To insert data into a relation, we either:
 - * specify a tuple to be inserted
 - * write a query whose result is a set of tuples to be inserted
- in relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$
 where r is a relation and E is a relational algebra expression.
- The insertion of a single tuple is expressed by letting E be a constant relation containing one tuple.



Insertion Examples: Review

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.

$$account \leftarrow account \cup \{("Perryridge", A-973, 1200)\}$$

$$depositor \leftarrow depositor \cup \{("Smith", A-973)\}$$

- Provide as a gift for all loan customers in the Perryridge branch, a \$200 savings account. Let the loan number serve as the account number for the new savings account.

$$r_1 \leftarrow (\sigma_{branch_name = "Perryridge"}(borrower \bowtie loan))$$

$$account \leftarrow account \cup \Pi_{branch_name, loan_number, 200}(r_1)$$

$$depositor \leftarrow depositor \cup \Pi_{customer_name, loan_number}(r_1)$$




Updating: Review

- A mechanism to change a value in a tuple without changing *all* values in the tuple
- Use the generalized projection operator to do this task

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_i, \dots, F_n}(r)$$

- Each F_i is either
 - * the i^{th} attribute of r , if the i^{th} attribute is not updated, or,
 - * if the attribute is to be updated F_i is an expression, involving only constants and the attributes of r , which gives the new value for the attribute



Update Examples: Review

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \Pi_{account_number, branch_name, balance * 1.05}(account)$$

- Pay all accounts with balances over \$10,000 6 percent interest and pay all others 5 percent

$$account \leftarrow \Pi_{account_number, branch_name, balance * 1.06}(\sigma_{BAL > 10000}(account)) \cup \Pi_{account_number, branch_name, balance * 1.05}(\sigma_{BAL \leq 10000}(account))$$





Create Table with Integrity Constraints: Review

- **not null**
- **primary key** (A_1, \dots, A_n)

Example: Declare *branch_name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

```
create table branch
(branch_name char(15),
 branch_city char(30),
 assets integer,
 primary key (branch_name))
```

primary key declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89



Drop and Alter Table Constructs: Review

- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation:

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

where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.

- * All tuples in the relation are assigned *null* as the value for the new attribute.

- The **alter table** command can also be used to drop attributes of a relation:

```
alter table r drop A
```

where *A* is the name of an attribute of relation *r*

- * Dropping of attributes not supported by many databases





Basic Query Structure: Review

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

```
select  $A_1, A_2, \dots, A_n$   
from  $r_1, r_2, \dots, r_m$   
where  $P$ 
```

- * A_i represents an attribute
- * R_i represents a relation
- * P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation.



Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

```
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  
         $account.branch\_name = 'Perryridge'$  )
```





Example Query

- Find all customers who have at least two accounts at the Perryridge branch.

```
select distinct T.customer_name
from depositor as 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')
```



Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the average account balance of those branches where the average account balance is greater than \$1200.

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

Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.





With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all accounts with the maximum balance

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



Complex Query using With Clause

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
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
```





Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

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

- Delete all accounts at every branch located in the city 'Needham'.

```
delete from account  
where branch_name in (select branch_name  
                        from branch  
                        where branch_city = 'Needham')
```



Example Query

- Delete the record of all accounts with balances below the average at the bank.

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

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:
 1. First, compute **avg** balance and find all tuples to delete
 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)





Modification of the Database – Insertion [1]

- Add a new tuple to *account*

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

or equivalently

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

- Add a new tuple to *account* with *balance* set to null

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



Modification of the Database – Insertion [2]

- Provide as a gift for all loan customers of the Perryridge branch, a \$200 savings account. Let the loan number serve as the account number for the new savings account

```
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
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like **insert into table1 select * from table1** would cause problems)





Modification of the Database – Updates

- Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.

- * Write two **update** statements:

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

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

- * The order is important
- * Can be done better using the **case** statement (next slide)



Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.

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





Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's loan number but has no need to see the loan amount. This person should see a relation described, in SQL, by

```
(select customer_name, loan_number
   from borrower, loan
   where borrower.loan_number = loan.loan_number )
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.



Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's loan number but has no need to see the loan amount. This person should see a relation described, in SQL, by

```
(select customer_name, loan_number
   from borrower, loan
   where borrower.loan_number = loan.loan_number )
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.





View Definition

- A view is defined using the **create view** statement which has the form
create view v as < query expression >
where <query expression> is any legal SQL expression. The view name is represented by *v*.
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
 - * Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.



Example Queries

- A view consisting of branches and their customers
create view all_customer as
(select branch_name, customer_name
from depositor, account
where depositor.account_number =
account.account_number)
union
(select branch_name, customer_name
from borrower, loan
where borrower.loan_number = loan.loan_number)
- Find all customers of the Perryridge branch
select customer_name
from all_customer
where branch_name = 'Perryridge'





Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.



View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:
 - 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 relations are present in e_1
- As long as the view definitions are not recursive, this loop will terminate





Update of a View

- Create a view of all loan data in the *loan* relation, hiding the *amount* attribute

```
create view branch_loan as  
    select branch_name, loan_number  
from loan
```

- Add a new tuple to *branch_loan*

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

This insertion must be represented by the insertion of the tuple ('L-307', 'Perryridge', *null*) into the *loan* relation



Updates Through Views (Cont.)

- Some updates through views are impossible to translate into updates on the database relations

```
* create view v as  
    select branch_name from account  
insert into v values ('L-99', 'Downtown', 23)
```

- Others cannot be translated uniquely

```
* insert into all_customer values ('Perryridge', 'John')
```

⇒ Have to choose loan or account, and create a new loan/account number!

- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation





Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form
create assertion <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
 - * This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting
for all $X, P(X)$
is achieved in a round-about fashion using
not exists X such that not $P(X)$



Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance of \$1000.00

```
create assertion balance_constraint check  
(not exists (  
  select *  
  from loan  
  where not exists (  
    select *  
    from borrower, depositor, account  
    where loan.loan_number = borrower.loan_number  
      and borrower.customer_name =  
        depositor.customer_name  
      and depositor.account_number =  
        account.account_number  
      and account.balance >= 1000)))
```





Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
create assertion sum_constraint check
  (not exists (select *
    from branch
    where (select sum(amount)
      from loan
      where loan.branch_name =
        branch.branch_name )
    >= (select sum (amount)
      from account
      where loan.branch_name =
        branch.branch_name )))
```



Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema (covered in Chapter 8):

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.





Authorization Specification in SQL

- The **grant** statement is used to confer authorization
grant <privilege list>
on <relation name or view name> **to** <user list>
- <user list> is:
 - * a user-id
 - * **public**, which allows all valid users the privilege granted
 - * A role (more on this in Chapter 8)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).



Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
 - * Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *branch* relation:
grant select on branch to U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges
- more in Chapter 8





Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.
revoke <privilege list>
on <relation name or view name> **from** <user list>
- Example:
revoke select on branch from U₁, U₂, U₃
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public**, all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.



Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded SQL*.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement > END_EXEC

Note: this varies by language (for example, the Java embedding uses

SQL { ... };)





Example Query

- From within a host language, find the names and cities of customers with more than the variable amount dollars in some account.

- Specify the query in SQL and declare a *cursor* for it
EXEC SQL

```
declare c cursor for  
select customer_name, customer_city  
from depositor, customer, account  
where depositor.customer_name = customer.customer_name  
       and depositor.account_number = account.account_number  
       and account.balance > :amount  
END_EXEC
```



Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated
EXEC SQL **open** c END_EXEC
- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.
EXEC SQL **fetch** c **into** :cn, :cc END_EXEC
Repeated calls to **fetch** get successive tuples in the query result
- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.
EXEC SQL **close** c END_EXEC

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.





Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

```
declare c cursor for  
select *  
from account  
where branch_name = 'Perryridge'  
for update
```

- To update tuple at the current location of cursor c

```
update account  
set balance = balance + 100  
where current of c
```



Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```
char * sqlprog = "update account  
set balance = balance * 1.05  
where account_number = ?"
```

```
EXEC SQL prepare dynprog from :sqlprog;
```

```
char account[10] = "A-101";
```

```
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

