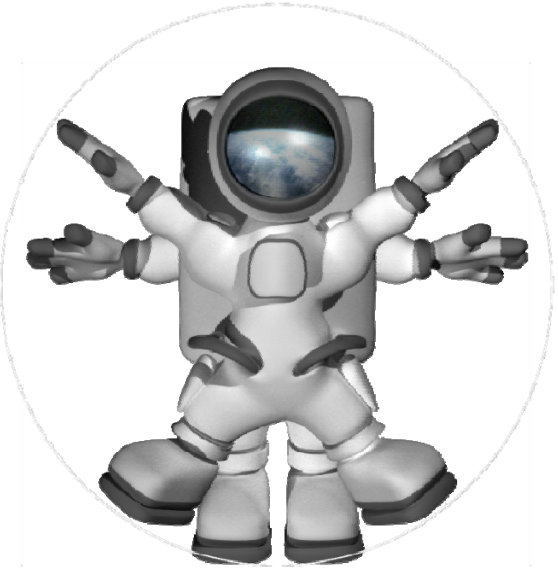
Chapter 4: Advanced SQL



Contents

- SQL Data Types and Schemas
- Integrity Constraints
- Authorization
- Trigger
- SQL Function
- Stored Procedure
- Embedded SQL

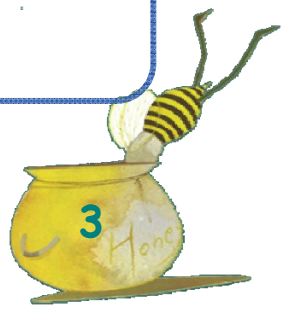
Built-in Data Types in SQL

- **date**: Made up of year-month-day in the format yyyy-mm-dd
- **time**: Made up of hour:minute:second in the format hh:mm:ss

date '2005-07-27'

time '09:00:30'

```
Insert into student (name, department, birthday)
values (smith, SE, date '2009-08-23')
```



Cont.

Built-in Data Types in SQL

- timestamp: date plus time of day
- interval: period of time
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

```
timestamp '2005-7-27 09:00:30'
```

```
interval '2 years 3 months'
```

```
update license
Set expireDay = expireDay + interval '2 years 6 months'
Where id ='12345'
```

Cont.

Build-in Data Types in SQL

- **Can get current date/time/timestamp**

```
CURRENT_TIME
CURRENT_DATE
CURRENT_TIMESTAMP
```

```
select CURRENT_TIME;
```

```
select CURRENT_DATE;
```

```
select CURRENT_TIMESTAMP;
```

Cont.

Build-in Data Types in SQL

- **Can extract values of individual fields from date/time/timestamp/interval**

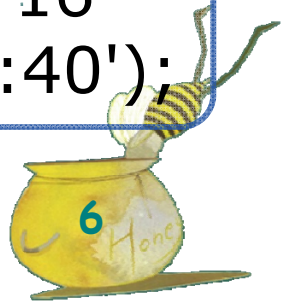
```
extract (year from r.starttime)
```

year, month, day, hour, minute, second

Return double precision value

```
select extract (day from timestamp '2001-02-16 20:38:40');
```

Result: 16



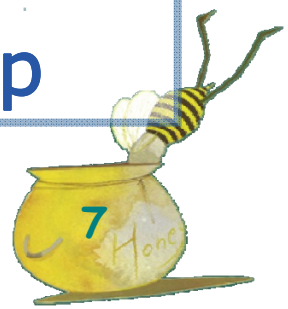
Cont.

Build-in Data Types in SQL

- **Can cast string types to date/time/ timestamp**

```
cast <string-valued-expression> as date
cast <string-valued-expression> as time
cast <string-valued-expression> as timestamp
```

```
cast '2008-2-22' as date
cast '13:22:34' as time
cast '2008-2-22 13:22:34' as timestamp
```



User-Defined Types

- **create type construct in SQL creates user-defined type**

```
create type Dollars as numeric (12,2) final
```

```
create type Ponds as numeric (12,2) final
```

```
create table account
(account_number  char(10),
 branch_name     char(15),
 balance         Dollars)
```

User-Defined Types Cont.

- **SQL apply strong type checking on user-defined types**

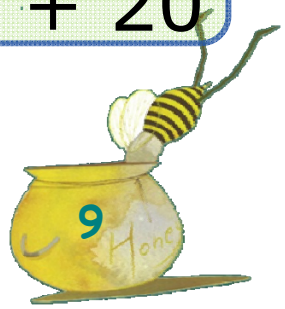
account.balance + 20 error

account.balance > 30 error

account.balance = 35 error

(cast *account.balance* to numeric(12,2)) > 30
+ 20

Account.balance = cast 35 to dollars



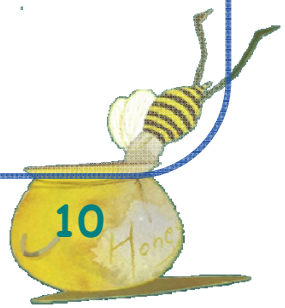
User-Defined Types Cont.

- **create domain** construct in SQL creates domain type

```
create domain sname as varchar(30) not null
```

The difference between Domain and Type:

- Domains can have constraints specified on them.
- Domains can be assigned to or compared with other domain type as long as the underlying types are compatible



Large-Object Types

- Photos, videos, CAD files, etc. are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of un-interpreted binary data
  - **clob**: character large object -- object is a large collection of character data
  - a **locator** is returned rather than the large object itself

Book_review clob(10KB)

Image blob(10MB)
Movie blob(10GB)

Integrity Constraints

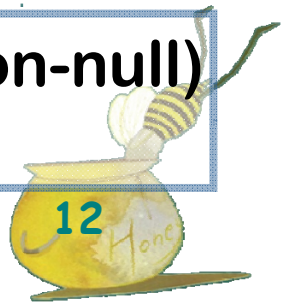
- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data integrity.

Example

A checking account must have a balance greater than $10,000.00

A salary of a bank employee must be at least $4.00 an hour

A customer must have a (non-null) phone number



Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraint.
  - test values inserted in the database
  - test queries to ensure that the comparisons make sense.

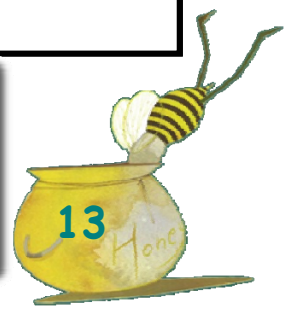
Not a meaningful query

Example

Find all customers who have the same name as branch

- To forbid this kind of query, customer_name and brach_name should have distinct domains

```
create type cnametype as char(30) final
create type bnametype as char(30) final
```



Constraints on a Single Relation

- not null
- primary key
- unique
- check (*P*), where *P* is a predicate

not null Constraint

- Declare *branch_name* for *branch* is **not null**
- Declare the domain *Dollars* to be **not null**

```
create domain Dollars numeric(12,2) not null
```

```
create table account
(account_number  char(10),
 branch_name     char(15) not null,
 balance         Dollars)
```

The unique Constraint

- **unique** ($A_1, A_2, \ldots, A_m$)
  - The unique specification states that the attributes

    $A_1, A_2, \ldots, A_m$

    form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

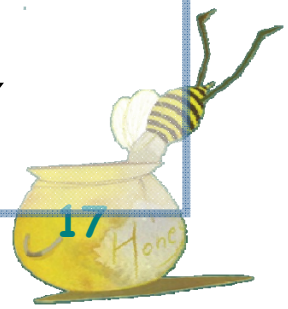
The check clause

- **check (*P*), where *P* is a predicate**

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),
         check (assets >= 0))
```



Cont.

The check clause

- The check clause in SQL-92 permits domains to be restricted:
  - Use check clause to ensure that an hourly_wage domain allows only values greater than a specified value.

```
create domain hourly_wage numeric(5,2)
    constraint value_test check(value >= 4.00)
```

Referential Integrity Cont.

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation

If "Perryridge" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch "Perryridge".

- Primary and candidate keys and foreign keys can be specified as part of the SQL create table statement

Referential Integrity

- The primary key clause lists attributes that comprise the primary key.
- The unique clause lists attributes that comprise a candidate key.
- The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key.
  - By default, a foreign key references the primary key attributes of the referenced table.

Example

```
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         numeric(12,2),
primary key (branch_name ))
```

Example

```
create table account
(account_number   char(10),
 branch_name      char(15),
 balance          integer,
 primary key (account_number),
 foreign key (branch_name) references branch )
```

```
create table depositor
(customer_name    char(20),
 account_number   char(10),
 primary key (customer_name, account_number),
 foreign key (account_number ) references account,
 foreign key (customer_name ) references customer )
```

Example

```
create table account
(account_number   char(10),
 branch_name      char(15),
 balance          integer,
 primary key (account_number),
 foreign key (branch_name) references branch
                  on delete cascade
                  on update cascade )
```

Cascade to account relation, deleting tuple that refers to the branch that was deleted

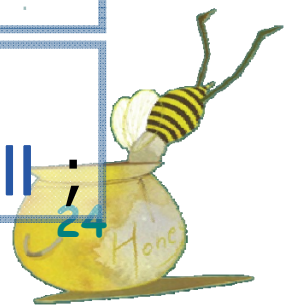
Updates field brach_name of referencing tuples in account to the new value

```
create table account
(account_number   char(10),
 branch_name      char(15),
 balance          integer)
```

```
alter table account
  add primary key (account_number),
  add constraint  fkey1
        foreign key(branch_name)  references branch,
  add constraint  ckbal  check (balance>10);
```

```
alter table account drop primary key;
alter table account drop constraint fkey1;
alter table account drop constraint ckbal;
```

```
alter table account
     alter column branch_name char(20) not null ;
```



```
create table couples
(name char(15),
 working_year integer,
 salary_account  char(30)
 spouse char(15),
 primary key (name),
 constraint fk1 foreign key (spouse)
                    references couple initially deferred );
```

constraint would be checked at the end of a transaction

```
create table couples
(name char(15),
 working_year integer,
 salary_account  char(30)
 spouse char(15),
 primary key (name),
 constraint fk1 foreign key (spouse)
                references couple deferrable );
```

Checked immediately by default, but can be deferred when desired

```
set constraints fk1 deferred ;
```

If used in a transaction, checking are deferred to the end of the transaction

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>
```

Assertions Cont.

- 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*)

SQL do not provide this construct directory

$$\forall x(P) \equiv \neg(\exists x(\neg P))$$

Example

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

$$\forall branch(T_{balacne} > T_{amount}) \equiv \neg(\exists branch(T_{balance} <= T_{amount}))$$

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

Example

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

owned by one of the loan's borrower

$$\forall loan(\exists account(balance >= 1000)) \equiv \neg(\exists loan(\neg(\exists account(balance >= 1000))))$$

owned by one of the loan's borrower

```
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) ) )
```

Loan

L_number	B_name	Amount
L-12	Perryridge	900
L-15	Round Hill	350

Borrower

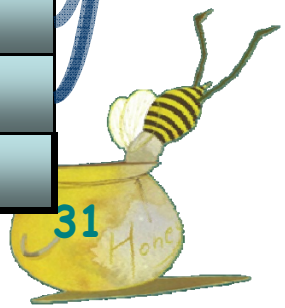
C-name	L_number
Hayes	L-15
Mike	L-12
Lisa	L-12
Smith	L-15

Depositor

C-name	A_number
Mike	A-101
Mike	A-151
Lisa	A-101
Smith	A-123

Account

A_number	B_name	Amount
A-151	Downtown	1500
A-101	Perryridge	900
A-123	Round Hill	2000



Loan

L_number	B_name	Amount
L-12	Perryridge	900
L-15	Round Hill	350

Borrower

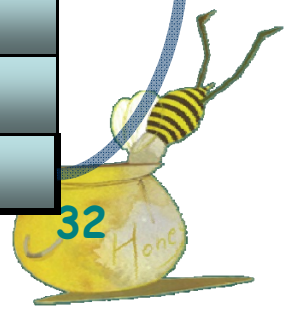
C-name	L_number
Hayes	L-15
Mike	L-12
Lisa	L-12
Smith	L-15

Depositor

C-name	A_number
Mike	A-101
Mike	A-151
Lisa	A-101
Smith	A-123

Account

A_number	B_name	Amount
A-151	Downtown	1500
A-101	Perryridge	900
A-123	Round Hill	2000



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.

Authorization

Forms of authorization to modify the database schema:

- 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.

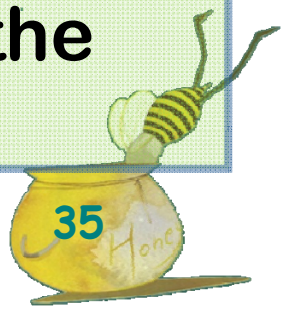
Authorization Specification in SQL

- The **grant** statement is used to confer authorization

A list of Privileges
All privilege, all allowable privileges

grant <privilege list>
on <relation name or view name> to <user list>

A list of user-id
public, all current and future users of the system



Cont.

Authorization Specification in SQL

- 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

Privileges in SQL

- **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

Revoking Authorization in SQL

- **The revoke statement is used to revoke authorization.**

revoke <privilege list>
on <relation name or view name>
from <user list>

revoke select on *branch* from U_1, U_2, U_3

Revoking Authorization in SQL

- 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

Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must: Event → Condition → Action
  - Specify when trigger is to be executed
  - Specify the actions to be taken when the trigger executes.

Example

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - setting the account balance to zero
  - creating a loan in the amount of the overdraft
  - giving this loan a loan number identical to the account number of the overdrawn account

Event: an update to the *account* relation

Condition: update results in a negative *balance* value.

Action: Let *t* denotes the *account* tuple with a negative *balance* value

1. Insert a new tuple s in the *loan* relation

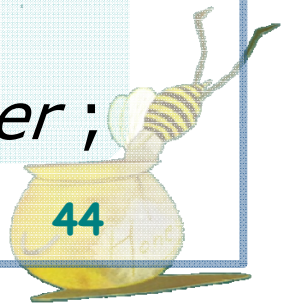
s[loan_number] = t[account_number]
s[branch_name] = t[branch_name]
s[amount] = - t[balance]

2. Insert a new tuple u in the *borrower* relation

u[customer_name] = the depositor of the account
u[loan_number] = t[account_number]

3. Set t[balance] to 0.

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    insert into loan
        values (nrow.account_number,
                    nrow.branch_name, – nrow.balance);
    insert into borrower
        (select customer_name, account_number
         from depositor
         where depositor.account_number
                        =nrow.account_number);
    update account set balance = 0
    where account.account_number
                        = nrow.account_number ;
end
```



■Account

A_number	B_name	Amount
A-151	Perryridge	-300
A-101	Perryridge	1900
A-123	Round Hill	2350
A-210	Uptown	1000

nrow

A-151	Perryridge	1500

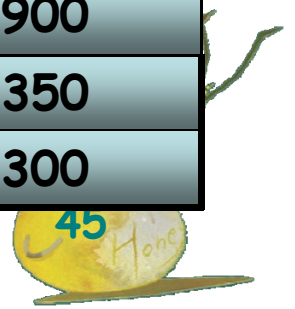
values (*nrow.account_number,*
nrow.branch_name,–nrow.balance);

A-151	Perryridge	300

Insert into

■Loan

L_number	B_name	Amount
L-11	Downtown	1500
L-12	Perryridge	900
L-15	Round Hill	350
A-151	Perryridge	300



Account

A_number	B_name	Amount
A-151	Perryridge	-300
A-101	Perryridge	1900
A-123	Round Hill	2350
A-210	Uptown	1000

nrow

A-151	Perryridge	1500

Depositor

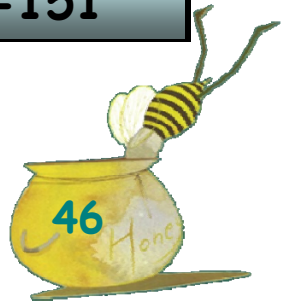
C-name	A_number
Hayes	A-101
Hayes	A-151
Lisa	A-101
Johnson	A-210
Smith	A-123

Insert into

Borrower

C-name	L_number
Hayes	L-11
Mike	L-15
Lisa	L-12
Lindsay	L-11
Hayes	A-151

C-name	A_number
Hayes	A-151



Account

A_number	B_name	Amount
A-151	Perryridge	0
A-101	Perryridge	1900
A-123	Round Hill	2350
A-210	Uptown	1000

A-151	Perryridge	1500

Borrower

C-name	L_number
Hayes	L-11
Mike	L-15
Lisa	L-12
Lindsay	L-11
Hayes	A-151

Loan

L_number	B_name	Amount
L-11	Downtown	1500
L-12	Perryridge	900
L-15	Round Hill	350
A-151	Perryridge	300

Triggers in SQL

- **Triggering event can be**

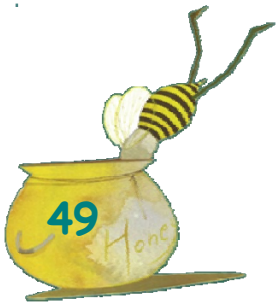
  insert, delete or update

- **Triggers on update can be restricted to specific attributes**

```
create trigger overdraft-trigger after update
of balance on account
```

Triggers in SQL Cont.

- Values of attributes before and after an update can be referenced
  - referencing old row as : for deletes and updates
  - referencing new row as: for inserts and updates



Triggers in SQL Cont.

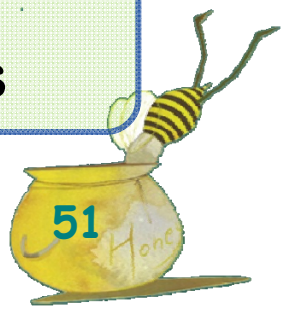
- **activated before an event, which can serve as extra constraints**

```
create trigger setnull-trigger before insert on r
referencing new row as nrow
for each row
when nrow.phone-number = ' '
set nrow.phone-number =null
```

Triggers in SQL Cont.

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use for each statement
  - Use referencing old table or referencing new table

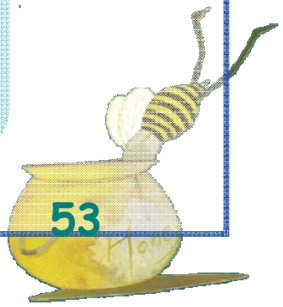
Can be more efficient when dealing with SQL statements that update a large number of rows



Example

- Suppose a warehouse has the following tables
  - *inventory*(*item, level*) : How much of each item is in the warehouse
  - *minlevel*(*item, level*) : What is the minimum desired level of each item
  - *reorder*(*item, amount*) : What quantity should we re-order at a time
  - *orders*(*item, amount*) : Orders to be placed (read by external process)

```
create trigger reorder-trigger after update of amount
 on inventory
 referencing old row as orow, new row as nrow
 for each row
 when nrow.level <=
          (select level
           from minlevel
           where minlevel.item = orow.item)
      and
      orow.level >
          (select level
          from minlevel
          where minlevel.item = orow.item)
 begin
     insert into orders
               (select item, amount
                from reorder
                where reorder.item = orow.item)
 end
```



inventory

Item	Level
R-201	890
B-101	1900
B-123	2390
D-210	1245

nrow

nrow

B-101	950
B-101	2206

orow

orow

minlevel

C-name	A_number
R-201	1000
B-101	2000
B-123	2300
D-210	1000

reorder

Item	Amount
R-201	500
B-101	700
B-123	400
D-210	200

Insert into

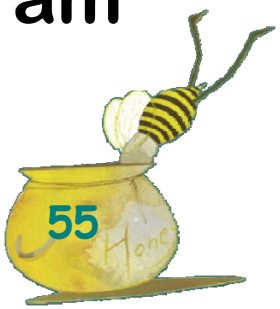
orders

Item	Amount
R-201	500
B-101	700

Item	Amount
B-101	700

Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external language
  - Stored in the database
  - Functions can be used in SQL queries
  - Procedure can be invoked either from SQL Procedure or from external program

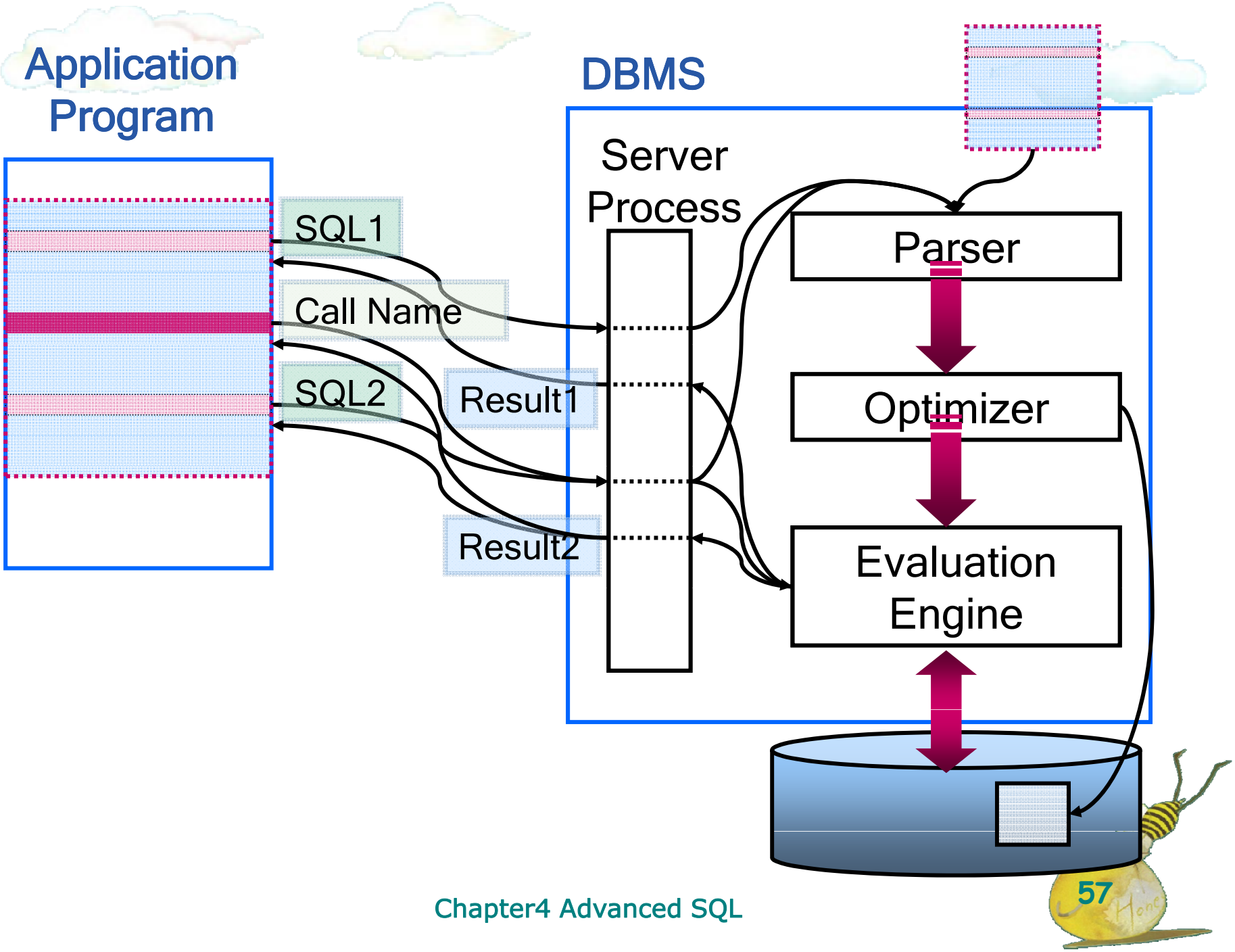


Cont.

Functions and Procedures

- SQL:1999 supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- SQL:2003 supports table-valued functions, which can return a relation as a result
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

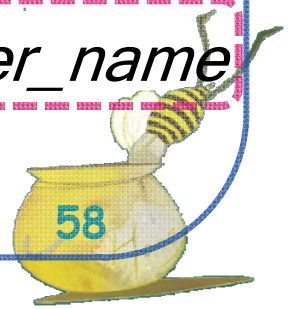
  Oracle: PL/SQL SQL Server: TransactSQL


Application Program
DBMS
Server Process
SQL1
Call Name
SQL2
Result1
Result2
Parser
Optimizer
Evaluation Engine

SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```
create function account_count
        (customer_name varchar(20))
returns integer
   begin
      declare a_count integer;
      select count (*) into a_count
      from depositor
      where depositor.customer_name
                                  = account_count.customer_name
       return a_count;
    end
```



SQL Functions Cont.

- **Find the name and address of each customer that has more than one account.**

```
select customer_name, customer_street, customer_city
from customer
where account_count (customer_name ) > 1
```

SQL:2003 added functions that return a relation as a result

- Return all accounts owned by a given customer

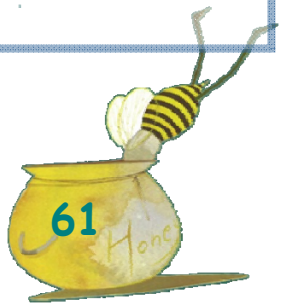
```
create function accounts_of (customer_name char(20))
    returns table ( account_number char(10),
                    branch_name char(15),
                    balance numeric(12,2) )
    return table
        ( select account_number, branch_name, balance
          from depositor D, account A
          where D.account_number = A.account_number
                and D.customer_name=
                                    accounts_of.customer_name )

select *
from table (accounts_of ('Smith'))
```

SQL Procedures

- The *account_count* function could instead be written as procedure

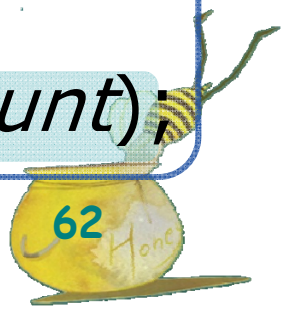
```
create procedure account_count_proc
    (in customer_name  varchar(20), out account integer)
     begin
          select count(account_number) into a_count
          from depositor
          where depositor.customer_name =
                         account_count_proc.customer_name
     end
```



SQL Procedures Cont.

- Procedures can be invoked either from an SQL procedure or from external program, using the call statement.
- SQL:1999 allows more than one function/procedure of the same name (called name overloading), as long as the number of arguments differ, or at least the types of the arguments differ

```
declare a_count integer;
call account_count_proc ('Smith', a_count);
```



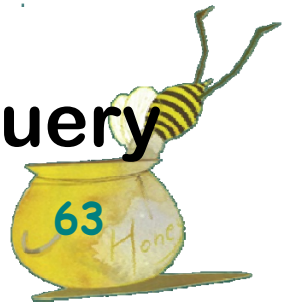
Procedural Constructs

- Compound statement: **begin** … **end**,
  - May contain multiple SQL statements between **begin** and **end**.
- **while** and **repeat** statements:

```
declare n integer default 0;
while n < 10 do
        set n = n + 1
end while
```

```
repeat
        set n = n – 1
until n = 0
end repeat
```

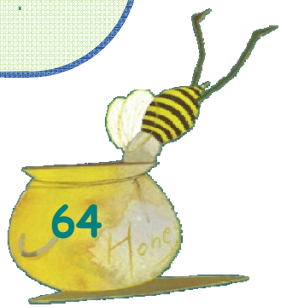
- **for** loop
  - Permits iteration over all results of a query



Procedural Constructs Cont.

- find the total of all account balances at the branch Perryridge

```
declare n integer default 0;
for r as
    select balance
    from account
    where branch_name = 'Perryridge'
do
    set n = n + r.balance
end for
```



Procedural Constructs Cont.

- **Conditional statements (if-then-else)**
  - **To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)**

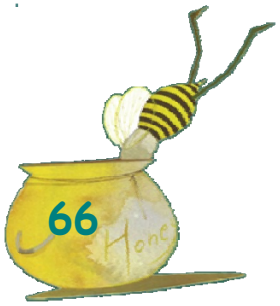
```
if r.balance < 1000
    then set l = l + r.balance
elseif r.balance < 5000
    then set m = m + r.balance
else set h = h + r.balance
end if
```

Cont.

Procedural Constructs

- **Signaling of exception conditions, and declaring handlers for exceptions**

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
    ...
      ..  signal out-of-stock
end
```



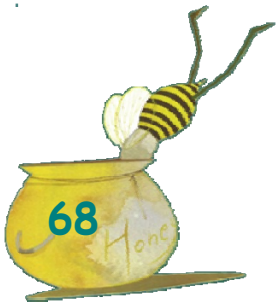
Queries

- Three ways to get the effect of a query:
  1. Queries producing one value can be the expression in an assignment.
  2. Single-row select . . . into.
  3. Cursors.

Assignment/Query

- If p is a local variable, we can get the number of accounts in the branch perryridge:

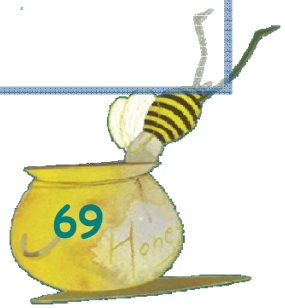
```
set p=(select count(account_number)
        from Account
        where branch_name = 'Perryridge');
```



select . . . into

- An equivalent way to get the value of a query that is guaranteed to return a single tuple is by placing into <variable> after the select clause.

```
declare p as integer
declare a as real
select count(account_number), avg(balance) into p, a
from Account
where branch_name = 'Perryridge';
```



Cursors

- A cursor is essentially a tuple-variable that ranges over all tuples in the result of some query.
- Declare a cursor *c* by:

```
declare c cursor for <query>;
```

Opening and Closing Cursors

- To use cursor *c*, we must issue the command:

```
open c;
```

The query of *c* is evaluated, and *c* is set to point to the position just before the first tuple of the result.

- When finished with *c*, issue command:

```
close c;
```

Fetching Tuples From a Cursor

- To get the next tuple from cursor c, issue command:

  fetch from c into $x_1, x_2,\ldots,x_n$;

  – The *x*'s are a list of variables, one for each component of the tuples referred to by *c*.

  – c is moved automatically to the next tuple.

Breaking Cursor Loops

- The usual way to use a cursor is to create a loop with a **fetch** statement, and do something with each tuple fetched.
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver.

Breaking Cursor Loops Cont.

- **Each SQL operation returns a *status*, which is a 5-digit number.**

```
00000 = "Everything is OK."
02000 = "Failed to find a tuple."
```

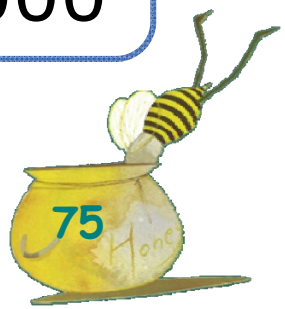
- **In SQL Procedure, we can get the value of the status in a variable called SQLSTATE.**

Cont.

Breaking Cursor Loops

- We may declare a condition, which is a boolean variable that is true if and only if SQLSTATE has a particular value.
- Example: We can declare condition NotFound to represent 02000 by:

```
declare NotFound condition for
                        SQLSTATE '02000'
```

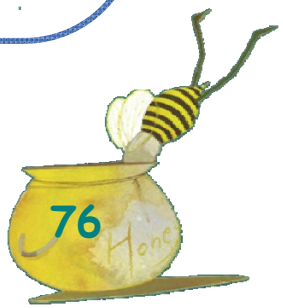


Breaking Cursor Loops

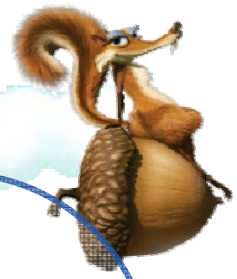
Cont.

- **The structure of a cursor loop is thus:**

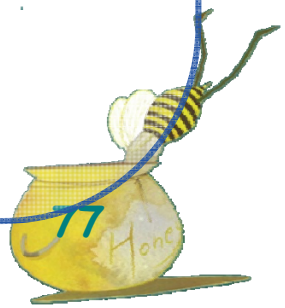
```
cursorLoop: loop
        …
        fetch c into … ;
        if NotFound then leave cursorLoop;
        end if;
        …
end loop;
```



Example



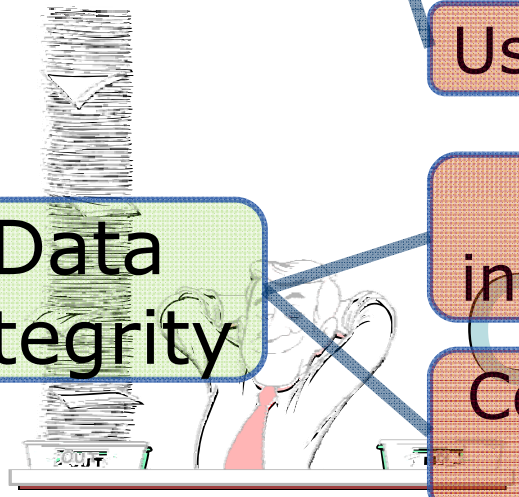
```
declare n  integer default 0;
declare bala  integer default 0;
declare NotFound condition for SQLSTATE '02000';
declare cursor c for
        (select balance from account
         where branch_name = 'Perryridge');
open c;
menuLoop: loop
    fetch c into bala;
    if NotFound then leave menuLoop end if;
    set n=n+bala
end loop;
close c;
```



Data types

- Date types: Date, Time, Timestamp, Interval; Current_Time(date/timestamp); Extract(year from r.birthday)
- Large object types: Blob, Clob;
- User defined types: create type, create domain;

Data integrity



- Single relation integrity constraints: Not null, check, unique, primary key, foreign key;
- Complex integrity constraints: create assertion *ast1* check(P)

Conclusions

Authorization: grant, revoke
➔ select, update, insert, delete, all privilege

trigger: create trigger tr1 before/after insert/delete/update of A1 on r1
referencing old row as / new row as
for each row
when p
begin
---
end

function and procedure

- create function
- create procedure
- procedural constructs
- access results of SQL

Questions?

End of Chapter 4