Set Comparison – “all” Clause

Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                  from instructor
                  where dept name = 'Biology');
```
### Definition of “all” Clause

- \( F \text{ <comp> all } r \iff \forall t \in r \ (F \text{ <comp> } t) \)

\[
\begin{array}{c|c}
5 & 0 \\
6 & 5 \\
\end{array}
\]

(5 < all 6) = false

\[
\begin{array}{c|c}
6 & 10 \\
\end{array}
\]

(5 < all 10) = true

\[
\begin{array}{c|c}
4 & 5 \\
\end{array}
\]

(5 = all 5) = false

\[
\begin{array}{c|c}
4 & 6 \\
\end{array}
\]

(5 ≠ all 6) = true (since 5 ≠ 4 and 5 ≠ 6)

\( (\neq \text{ all}) \equiv \not \in \)

However, \( (= \text{ all}) \neq \in \)

### Test for Empty Relations

- The \textit{exists} construct returns the value \textbf{true} if the argument subquery is nonempty.
- \textbf{exists} \( r \iff r \neq \emptyset \)
- \textbf{not exists} \( r \iff r = \emptyset \)
Use of “exists” Clause

• Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

\[
\begin{align*}
\text{select} & \quad \text{course_id} \\
\text{from} & \quad \text{section as} \ S \\
\text{where} & \quad \text{semester = 'Fall' and year = 2009 and} \\
& \hspace{1em} \text{exists (select * from section as} \ T \\
& \hspace{2em} \text{where semester = 'Spring' and year = 2010} \\
& \hspace{3em} \text{and S.course_id = T.course_id)};
\end{align*}
\]

• Correlation name – variable S in the outer query
• Correlated subquery – the inner query

Use of “not exists” Clause

• Find all students who have taken all courses offered in the Biology department.

\[
\begin{align*}
\text{select distinct} & \quad S.ID, S.name \\
\text{from} & \quad \text{student as} \ S \\
\text{where not exists} ( & \quad ( \text{select} \ \text{course_id} \\
& \hspace{1em} \text{from} \ \text{course} \\
& \hspace{2em} \text{where} \ \text{dept_name = 'Biology'}) \\
& \hspace{1em} \text{except} \\
& \hspace{2em} ( \text{select} \ T.course_id \\
& \hspace{3em} \text{from} \ \text{takes as} \ T \\
& \hspace{4em} \text{where} \ S.ID = T.ID));
\end{align*}
\]

• First nested query lists all courses offered in Biology
• Second nested query lists all courses a particular student took

• Note that \( X - Y = \emptyset \iff X \subseteq Y \)
• Note: Cannot write this query using = all and its variants
Test for Absence of Duplicate Tuples

- The unique construct tests whether a subquery has any duplicate tuples in its result.
- The unique construct evaluates to “true” if a given subquery contains no duplicates.
- Find all courses that were offered at most once in 2009
  
  ```sql
  select T.course_id
  from course as T
  where unique (select R.course_id
                  from section as R
                  where T.course_id = R.course_id
                  and R.year = 2009);
  ```

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the from clause
- Find the average instructors’ salaries of those departments where the average salary is greater than $42,000.
  
  ```sql
  select dept_name, avg_salary
  from (select dept_name, avg (salary) as avg_salary
           from instructor
           group by dept_name)
  where avg_salary > 42000;
  ```

- Note that we do not need to use the having clause
- Another way to write above query
  
  ```sql
  select dept_name, avg_salary
  from (select dept_name, avg (salary) as avg_salary
           from instructor
           group by dept_name) as dept_avg (dept_name, avg_salary)
  where avg_salary > 42000;
  ```
With Clause

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

\[
\text{with max\_budget (value) as} \\
(\text{select max(budget)} \\
\text{from department}) \\
\text{select department.name} \\
\text{from department, max\_budget} \\
\text{where department.budget = max\_budget.value};
\]

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

\[
\text{with dept\_total (dept\_name, value) as} \\
(\text{select dept\_name, sum(salary)} \\
\text{from instructor} \\
\text{group by dept\_name}), \\
dep\_total\_avg(value) as \\
(\text{select avg(value)} \\
\text{from dept\_total}) \\
\text{select dept\_name} \\
\text{from dept\_total, dept\_total\_avg} \\
\text{where dept\_total.value > dept\_total\_avg.value};
\]
Scalar Subquery

- Scalar subquery is one which is used where a single value is expected.
- List all departments along with the number of instructors in each department.
  
  ```sql
  select dept_name,
         (select count(*)
          from instructor
          where department.dept_name = instructor.dept_name)
  as num_instructors
  from department;
  ```

- Runtime error if subquery returns more than one result tuple.

Outerjoin

The normal join can “lose” information, because a tuple that doesn’t join with any from the other relation (dangles) has no vestage in the join result.

- The null value \(\perp\) can be used to “pad” dangling tuples so they appear in the join.
- Gives us the outerjoin operator \(\bowtie\).
- Variations: theta-outerjoin, left- and right-outerjoin (pad only dangling tuples from the left (respectively, right).
Join operations – Example

- **Relation course**

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Relation prereq**

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

- Observe that
  prereq information is missing for CS-315 and
course information is missing for CS-437

Left Outer Join

- **course natural left outer join prereq**

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
</tbody>
</table>
Right Outer Join

- course natural right outer join prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

Joined Relations

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the from clause
- Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<table>
<thead>
<tr>
<th>Join types</th>
<th>Join Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner join</td>
<td>natural</td>
</tr>
<tr>
<td>left outer join</td>
<td>on &lt;predicate&gt;</td>
</tr>
<tr>
<td>right outer join</td>
<td>using ( A_1, A_1, \ldots, A_n )</td>
</tr>
<tr>
<td>full outer join</td>
<td></td>
</tr>
</tbody>
</table>
Full Outer Join

- course natural full outer join prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

Is outerjoin operation (∇) fundamental?
A. Yes, it is a fundamental relational operation
B. No, it can be written using other relational operations

“Constructing” outerjoin

- Cross-product
  - Select (σ) on join attributes being equal
  - Add in (U) the “missing” items
- Can choose either left or right outerjoin
- Same in SQL
  - select *
    from left, right
    where left.id = right.id or left.id
  UNION
  select *
  from left
  where left.id not in (select id from right)
  UNION
  select *
  from left
  where right.id not in (select id from left)
Join operations – Example

- Relation course

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
</tr>
</tbody>
</table>

- Relation prereq

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

- Observe that
  - prereq information is missing for CS-315
  - course information is missing for CS-437

Joined Relations – Examples

- course inner join prereq on
  
  `course.course_id = prereq.course_id`

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
<td>BIO-301</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
<td>CS-190</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What is the difference between the above, and a natural join?

- course left outer join prereq on
  
  `course.course_id = prereq.course_id`

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
<td>BIO-301</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
<td>CS-190</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>CS-437</td>
<td></td>
<td></td>
<td>null</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Joined Relations – Examples**

- *course natural right outer join* `prereq`

```
<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>
```

- *course full outer join* `prereq using (course_id)`

```
<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>
```

**Extended (“Nonclassical”) Relational Algebra**

Adds features needed for SQL, bags.

1. Duplicate-elimination operator \( \delta \).
2. Extended projection.
3. Sorting operator \( \tau \).
4. Grouping-and-aggregation operator \( \gamma \).
5. Outerjoin operator \( \bowtie \).
Accessing SQL From a Programming Language

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- Various tools:
  - JDBC (Java Database Connectivity) works with Java
  - ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic. Other API's such as ADO.NET sit on top of ODBC
  - Embedded SQL
Key concept: 
**Cursor**

- Query returns a table
  - Could be viewed as a “Set” data type
  - Not all programming languages deal with this
- Instead, idea of a *cursor* to iterate over table
  - Access one row of result at a time
  - Typically used in a loop construct in the language
- Query processor “understands” cursor
  - Can start making results available before query completes

---

**JDBC**

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors
ODBC

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    † open a connection with a database,
    † send queries and updates,
    † get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- 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/1.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor
  ```
  EXEC SQL <embedded SQL statement >;
  ```
  Note: this varies by language:
  - In some languages, like COBOL, the semicolon is replaced with **END-EXEC**
  - In Java embedding uses  ```# SQL { .... }```
Embedded SQL (Cont.)

- Before executing any SQL statements, the program must first connect to the database. This is done using:
  
  ```sql
  EXEC SQL connect to  server user user-name using password;
  ```
  
  Here, server identifies the server to which a connection is to be established.

- Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., `:credit_amount`).

- Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

  ```sql
  EXEC-SQL BEGIN DECLARE SECTION
  int credit-amount;
  EXEC-SQL END DECLARE SECTION;
  ```

Embedded SQL (Cont.)

- To write an embedded SQL query, we use the `declare c cursor for <SQL query>` statement. The variable `c` is used to identify the query.

- Example:
  
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable `credit_amount` in the host language.
  
  - Specify the query in SQL as follows:

  ```sql
  EXEC SQL
  declare c cursor for
  select ID, name
  from student
  where tot_cred > :credit_amount
  END_EXEC
  ```
Embedded SQL (Cont.)

Example:

- From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable `credit_amount` in the host language.

Specify the query in SQL as follows:

```
EXEC SQL
  declare c cursor for
  select ID, name
  from student
  where tot_cred > :credit_amount
END_EXEC
```

- The variable `c` (used in the cursor declaration) is used to identify the query.

Embedded SQL (Cont.)

- The `open` statement for our example is as follows:

  ```
  EXEC SQL open c;
  ```

  This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable `credit-amount` at the time the `open` statement is executed.

- 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 :si, :sn END_EXEC
  ```

  Repeated calls to fetch get successive tuples in the query result.
Embedded SQL (Cont.)

- 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.

```sql
EXEC SQL close c;
```

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

Updates Through Embedded SQL

- Embedded SQL expressions for database modification (update, insert, and delete)
- Can update tuples fetched by cursor by declaring that the cursor is for update

```sql
EXEC SQL
    declare c cursor for
    select *
    from instructor
    where dept_name = 'Music'
    for update
```

- We then iterate through the tuples by performing fetch operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

```sql
update instructor
set salary = salary + 1000
where current of c
```
Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java).
  - Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects.
    - Example: functions to check if polygons overlap, or to compare images for similarity.
  - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

  ```sql
  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
  ```

- The function `dept_count` can be used to find the department names and budget of all departments with more that 12 instructors.

  ```sql
  select dept_name, budget
  from department
  where dept_count (dept_name) > 12
  ```
SQL functions (Cont.)

- Compound statement: `begin ... end`
  - May contain multiple SQL statements between `begin` and `end`.
- `returns` -- indicates the variable-type that is returned (e.g., integer)
- `return` -- specifies the values that are to be returned as result of invoking the function
- SQL function are in fact parameterized views that generalize the regular notion of views by allowing parameters.

Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all instructors in a given department

```sql
create function instructor_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 = instructor_of.dept_name)
```

- Usage

```sql
select *
from table (instructor_of('Music'))
```
The `dept_count` function could instead be written as procedure:

```sql
create procedure dept_count_proc(in dept_name varchar(20),
                                  out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the `call` statement.

```sql
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

Procedures and functions can be invoked also from dynamic SQL.

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.

---

**Language Constructs for Procedures & Functions**

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.

- Compound statement: `begin ... end`,
  - May contain multiple SQL statements between `begin` and `end`.
  - Local variables can be declared within a compound statements

- `While` and `repeat` statements:
  - `while boolean expression do
    sequence of statements;
  end while`

  - `repeat
    sequence of statements;
  until boolean expression
  end repeat`
Language Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
  - Example: Find the budget of all departments
    
    ```sql
    declare n integer default 0;
    for r as
    select budget from department
    do
    set n = n + r.budget
    end for
    ```

Language Constructs (Cont.)

- Conditional statements (**if-then-else**)
  - SQL:1999 also supports a **case** statement similar to C case statement
  - Example procedure: registers student after ensuring classroom capacity is not exceeded
    - Returns 0 on success and -1 if capacity is exceeded
    - See book (page 177) for details
  - Signaling of exception conditions, and declaring handlers for exceptions
    ```sql
    declare out_of_classroom_seats condition
    declare exit handler for out_of_classroom_seats
    begin
    ...
    .. signal out_of_classroom_seats
    end
    ```
    - The handler here is **exit** -- causes enclosing **begin..end** to be exited
    - Other actions possible on exception
External Language Routines

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```sql
create procedure dept_count_proc(
    in dept_name varchar(20),
    out count integer
) language C
external name 'usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name 'usr/avi/bin/dept_count'
```

External Language Routines

- SQL:1999 allows the definition of procedures in an imperative programming language, (Java, C#, C or C++) which can be invoked from SQL queries.
- Functions defined in this fashion can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.
- Declaring external language procedures and functions

```sql
create procedure dept_count_proc(
    in dept_name varchar(20),
    out count integer
) language C
external name 'usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name 'usr/avi/bin/dept_count'
```
External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power.
- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space.
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance.
  - Direct execution in the database system’s space is used when efficiency is more important than security.

Security with External Language Routines

- To deal with security problems, we can do one of the following:
  - Use sandbox techniques
    - That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code.
  - Run external language functions/procedures in a separate process, with no access to the database process’ memory.
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space.