Introduction to SQL

- Structured Query Language ('Sequel')
  - Serves as DDL as well as DML

- Declarative
  - Say what you want without specifying how to do it
  - One of the main reasons for commercial success of DBMSs

- Many standards and implementations
  - ANSI SQL
  - SQL-92/SQL-2 (Null operations, Outerjoins etc.)
  - SQL3 (Recursion, Triggers, Objects)
  - Vendor specific implementations

- “Bag Semantics” instead of “Set Semantics”
  - Used in commercial RDBMSs

Example:

- Create a Relation/Table in SQL

  ```sql
  CREATE TABLE Students
  (sid CHAR(9),
   name VARCHAR(20),
   login CHAR(8),
   age INTEGER,
   gpa REAL);
  ```

- Support for Basic Data Types
  - CHAR(n)
  - VARCHAR(n)
  - BIT(n)
  - BIT VARYING(n)
  - INT/INTEGER
  - FLOAT
  - REAL, DOUBLE PRECISION
  - DECIMAL(p,d)
  - DATE, TIME etc.
More Examples

□ And one for Courses

    CREATE TABLE Courses
    (courseid CHAR(6),
     department CHAR(20));

□ And one for their relationship!

    CREATE TABLE takes
    (sid CHAR(9),
     courseid CHAR(6));

□ Why?

□ Can also provide default values

    CREATE TABLE Students
    (sid CHAR(9),
     ...,
     age INTEGER DEFAULT 21,
     gpa REAL);

Examples Contd.

□ DATE and TIME
    □ Implementations vary widely
    □ Typically treated as strings of a special form
    □ Allows comparisons of an ordinal nature (<, > etc.)

□ DATE Example
    □ ‘1999-03-03’ (No Y2K problems)

□ TIME Examples
    □ ‘15:30:29’
    □ ‘15:30:29.3875’

□ Deleting a Relation/Table in SQL

    DROP TABLE Students;
Modifying Relation Schemas

- ‘Drop’ an attribute (column)
  
  ```sql
  ALTER TABLE Students DROP login;
  ```

- ‘Add’ an attribute (column)
  
  ```sql
  ALTER TABLE Students ADD phone CHAR(7);
  ```

- What happens to the new entry for the old records?
  - Default is ‘NULL’ or say
  
  ```sql
  ALTER TABLE Students ADD phone CHAR(7)  
  DEFAULT ‘unknown’;
  ```

- Always begin with ‘ALTER TABLE <TABLE_Name>’

- Can use DEFAULT even with regular definition (as in Slide 69)

---

How do you enter/modify data?

- INSERT command
  
  ```sql
  INSERT  
  INTO Students  
  VALUES (‘53688’,’Mark’,’mark2345’,23,3.9)
  ```

- Cumbersome (use bulk loading; described later)

- DELETE command
  
  ```sql
  DELETE  
  FROM Students S  
  WHERE S.name = ‘Smith’
  ```

- UPDATE command
  
  ```sql
  UPDATE Students S  
  SET S.age=S.age+1, S.gpa=S.gpa-1  
  WHERE S.sid = ‘53688’
  ```
Domains

- Domains: Similar to Structs and other user-defined types

  ```sql
  CREATE DOMAIN Email AS CHAR(8) DEFAULT 'unknown';
  ...
  login Email  // instead of login CHAR(8) DEFAULT 'unknown'
  ```

- Advantages: can be reused

  ```sql
  junkaddress Email,
  fromaddress Email,
  toaddress Email,
  ...
  ```

- Can DROP DOMAINS too!

  ```sql
  DROP DOMAIN Email;
  ```

  - Affects only future declarations

---

Keys

- To Specify Keys
  - Use PRIMARY KEY or UNIQUE
  - Declare alongside attribute
  - For multiattribute keys, declare as a separate line

  ```sql
  CREATE TABLE takes
  ( sid CHAR(9),
    courseid CHAR(6),
    PRIMARY KEY (sid,courseid)
  );
  ```

- What's the difference between PRIMARY KEY and UNIQUE?
  - Typically only one PRIMARY KEY but any number of UNIQUE keys
  - Implementor allowed to attach special significance
Creating Indices/Indexes

- **Why?**
  - Speeds up query processing time

- **For Students**
  
  ```sql
  CREATE INDEX indexone ON Students(sid);
  CREATE INDEX indextwo ON Students(login);
  ```

- **How to decide attributes to place indices on?**
  - One is (typically) created by default on PRIMARY KEY
  - Creation of indices on UNIQUE attributes is implementation-dependent
  - In general, physical database design/tuning is very difficult!
  - Use Tools: Microsoft SQLServer has an index selection Wizard

- **Why not place indices on all attributes?**
  - Too cumbersome for insertions/deletions/updates

- Like all things in computer science, there is a tradeoff! :-)

Other Properties

- ‘NOT NULL’ instead of DEFAULT

  ```sql
  CREATE TABLE Students
  (sid CHAR(9),
   name VARCHAR(20),
   login CHAR(8),
   age INTEGER,
   gpa REAL);
  ```

- Can insert a tuple without a value for gpa
  - NULL will be inserted

- If we had specified

  ```sql
  gpa REAL NOT NULL);
  ```

  - Insert cannot be made!
Module 2
Querying and Manipulations in the Relational Model *

(for use with CS5614)

In this module, we will study three different query languages/representations used in conjunction with the relational model. The first is relational algebra, an algebraic and procedural way for creating new relations from given ones. The second is Datalog, which is logical and declarative in nature (students familiar with the PROLOG programming language will find this all too natural). In fact, there are close correspondences between database systems and PROLOG. PROLOG can be thought of as a database system where all the data fits into main memory. In contrast, a distinguishing feature of RDBMSs is that they operate on secondary storage. Other than that (and some differences in query processing), there are excellent analogs to both cultures. Both PROLOG and RDBMSs are declarative. What we know to be relations are referred to as ‘predicates’ in PROLOG. A tuple is called a ground fact in PROLOG. A table is called an ‘extensional definition’ in PROLOG and so on. (Do not get bogged down by these specifics; we include them here just so that you can make the connection, if you are already familiar with PROLOG. Else, nothing to worry.) The third query representation is, of course, SQL that was introduced earlier. In the remainder of this document, we will introduce basic operations and manipulations that we can perform on relations. For each such basic operation, we will show how it is represented in each of the three different notations.

1. **Union of Relations:** The union of two relations \( R \) and \( S \) is the set of elements that are in \( R \) or in \( S \) or both. We assume that the schemas of \( R \) and \( S \) are alike (of course) and that their columns are also ordered alike (of course, again). We now give the three representations of the union relation:

   - \( R \cup S \) (simple, right?)
   - \( T(x,y,z,w) \leftarrow R(x,y,z,w) \).
     \( T(x,y,z,w) \leftarrow S(x,y,z,w) \).

   Notice that the variables \( x,y,z,w \) are merely ‘placeholders’ used for pattern matching; we could have written the above two as:

   \( T(a,b,c,d) \leftarrow R(a,b,c,d) \).
   \( T(e,f,g,h) \leftarrow S(e,f,g,h) \).

   - (SELECT *)

*No brownie points for guessing why we switched to a different document style for Module 2!
2. **Difference of Relations**: The difference \( R - S \) of two relations \( R \) and \( S \) is the set of elements that are in \( R \) but not in \( S \). As usual, we assume that the schemas of \( R \) and \( S \) are alike and that their columns are also ordered alike. Moreover, notice that \( R - S \) is not (generally) the same as \( S - R \).

- \( R - S \)
- \( T(x, y, z, w) \leftarrow R(x, y, z, w), \text{ NOT } S(x, y, z, w) \).
- (SELECT *
  FROM R)
  EXCEPT
  (SELECT *
  FROM S)

3. **Intersection of Relations**: The intersection \( R \cap S \) of two relations \( R \) and \( S \) is the set of elements that are in both \( R \) and \( S \). Again, we assume that the schemas of \( R \) and \( S \) are alike and that their columns are also ordered alike. Notice that \( R \cap S = R - (R - S) \).

- \( R \cap S \)
- \( T(x, y, z, w) \leftarrow R(x, y, z, w), S(x, y, z, w) \).
- (SELECT *
  FROM R)
  INTERSECT
  (SELECT *
  FROM S)

4. **Projection**: Operates on a single relation and removes some of the columns. Useful for restricting information. Assume that we want only the name and address from relation \( R \).

- \( \pi_{\text{name, address}} R \)
- \( T(x, y) \leftarrow R(x, y, z, w) \).
  Thus \( z, w \) become irrelevant attributes. We could instead reinforce this by writing
  \( T(x, y) \leftarrow R(x, y, -, -) \).
- SELECT name, address
  FROM R

5. **Selection**: Operates on a single relation and removes some of the rows. The removal is based on some condition specified by the user. For example, suppose we want all the tuples from \( R \) where the name is ‘Michael’.
\( \sigma_{\text{name}='Michael'} R \)
\( T(x,y,z,w) \leftarrow R(x,y,z,w), x='Michael'. \)

- SELECT *
  FROM R
  WHERE name = 'Michael'

6. **More Selections:** Selections become complicated when the conditions get longer, particularly with Datalog (the other two forms are pretty straightforward). Consider for example, when we want all the tuples from R where the name is ‘Michael’ OR when the gender is ‘M’. We write this in Datalog as:

\[
T(x,y,z,w) \leftarrow R(x,y,z,w), x='Michael'.
T(x,y,z,w) \leftarrow R(x,y,z,w), z='M'.
\]

Notice that we ‘split’ the condition across two rules, just as we do in the union case (Come to think of it, the OR is indeed the union of two conditions). Similarly, the comma in each of the above datalog rules models the AND condition. Let’s consider a more complicated condition: Select all the tuples from R that are neither male nor have the name ‘Fox’.

\[
T(x,y,z,w) \leftarrow R(x,y,z,w), z <> 'M', x <> 'Fox'.
\]

While selecting the tuples from R that are not both male and have the name ‘Fox’ is achieved by (why?):

\[
T(x,y,z,w) \leftarrow R(x,y,z,w), z <> 'M'.
T(x,y,z,w) \leftarrow R(x,y,z,w), x <> 'Fox'.
\]

7. **Cartesian Product:** This is the set of ‘pairs’ formed by choosing the first element from \( R \) and the second element from \( S \). In case of confusions among attribute names, disambiguate them by prefixing them with the relation name. The cartesian product of two relations \( R \) and \( S \) is given by:

- \( R \times S \)
- \( T(x_1,y_1,z_1,w_1,x_2,y_2,z_2,w_2) \leftarrow R(x_1,y_1,z_1,w_1), S(x_2,y_2,z_2,w_2). \)

- SELECT R.name, R.address, R.gender, R.birthday, S.name, S.address, S.gender, S.birthday
  FROM R,S

Notice how we disambiguate attributes in the SQL version. Also notice that if \( R \) has \( m \) tuples and \( S \) has \( n \) tuples, then \( R \times S \) will have \( mn \) tuples.
8. **Theta-Join**: This is just like the cartesian product but goes a step further. After forming the $mn$ tuples, it selects only a subset of them to include in its answer, based on some condition. Thus, the theta-join of two relations has the same number of columns as the cartesian product but not necessarily the same number of rows (some of the rows will be removed because they dissatisfy some condition). Consider for example, that we want to find ‘pairs’ of students such that the first person in the pair is always ‘Michael’. We get:

- $R \bowtie_{\text{R.name}='Michael'} S$
- $T(x_1,y_1,z_1,w_1,x_2,y_2,z_2,w_2) \leftarrow R(x_1,y_1,z_1,w_1), S(x_2,y_2,z_2,w_2), x_1='Michael'$.  
- $\text{SELECT } \text{R.name, R.address, R.gender, R.birthdate, S.name, S.address, S.gender, S.birthdate FROM R,S WHERE R.name = 'Michael'}$

Notice that we introduce the ‘bowtie’ symbol $\bowtie$ suffix-ed by the condition for indicating the theta-join. Also, realize that

$$R \bowtie_C S = \sigma_C(R \times S)$$

9. **Natural Join**: This is just a cleverer way to combine two relations into one. The basic idea is that if the two relations have some column(s) in common, then we can ‘collapse’ them into the same column in the final output. Moreover, we can do this only if the two tuples from the two relations agree in those common columns. Thus, it is similar to the cartesian product, but we ‘join’ only those pairs that match in their common attributes. Consider that we want to find the name, address, gender, gpa and birthdate of students in a single relation. Notice that gpa is available from $L$ but the other four attributes are present in $R$. So, we need a way to intelligently combine these two relations:

- $R \bowtie L$
- $T(x,y,z,v,w) \leftarrow R(x,y,z,w), L(x,y,v)$.
- $\text{SELECT } \text{R.name, R.address, R.gender, L.gpa, R.birthdate FROM R,L WHERE R.name=L.name AND R.address = L.address}$

10. **Renaming**: This is just a cool thing, in case we have too many naming conflicts and confusions arising. We can use this operator to rename a relation’s name and/or one or more of its attributes. For example, assume we want to rename $R$ to $M$ and make its columns to be called $m_1$, $m_2$ and $m_3$. This is most useful with relational algebra, like so:

- $\rho_{M(m_1,m_2,m_3)}(R)$
What is still to be covered
(and will be)

- Declaring constraints
  - Domain Constraints
  - Referential Integrity (Foreign Keys)

- More SQL Stuff
  - Subqueries
  - Aggregation

- SQL Peculiarities
  - Strange Phenomena
  - More on Bag Semantics
  - Ifs and Buts

- Embedding SQL in a Programming Environment
  - Accessing DBs from within a PL
  - (will be covered in Module 3)

What will be mentioned
(but not covered in detail)

- Triggers
  - Read Cow Book or Boat Book

- More SQL Gory Details

- Recursive Queries (SQL3)
  - Why do we need these?

- Security

- Authorization and Privacy

- Trends towards Object Oriented DBMSs
Tuple-Based Domain Constraints

- Already Seen
  - NOT NULL
  - UNIQUE, PRIMARY KEY etc.

- In General

```sql
CREATE TABLE Students
(sid CHAR(9),
 name VARCHAR(20),
 login CHAR(8),
 age INTEGER,
 gpa REAL,
 CHECK (gpa >= 0.0)
);
```

- Note: Implementations vary, but this is the general idea

- Other Complicated Forms
  - Constraints on whole relations, Assertions

---

Referential Integrity Constraints

- Foreign Keys
  - An attribute a of R1 is a foreign key if it “references” the primary key (say b) of another relation R2
  - In addition, there is a ref. integrity constraint from R1 to R2.

- Example
  - login is a FOREIGN KEY for Students

```sql
CREATE TABLE Students
(sid CHAR(9) PRIMARY KEY,
 name VARCHAR(20),
 login CHAR(8)
 REFERENCES Accounts(acct),
 age INTEGER,
 gpa REAL
);

CREATE TABLE Accounts
(
 acct CHAR(8) PRIMARY KEY
);
```
Alternatively

- Can use “FOREIGN KEY” construct

```sql
CREATE TABLE Students
(sid CHAR(9) PRIMARY KEY,
 name VARCHAR(20),
 login CHAR(8),
 age INTEGER,
 gpa REAL,
 FOREIGN KEY login
 REFERENCES Accounts(acct)
);

CREATE TABLE Accounts
(acct CHAR(8) PRIMARY KEY
);
```

- Note: acct should be declared as PRIMARY KEY for Accounts!
  - in both cases

---

### SQL Subqueries

- Given
  ```sql
  Students(sid,name,login,age,gpa)
  HasCar(sid,carname)
  ```

- Find
  - the car of the student with login="mark"

- Traditional Way

```sql
SELECT carname
FROM Students, HasCar
WHERE Students.login='mark'
AND Students.sid=HasCar.sid;
```

- The ‘Subway’

```sql
SELECT carname
FROM HasCar
WHERE sid=
  (SELECT sid FROM Students
   WHERE login='mark');
```
Aggregation

- **Given**
  \[
  \text{Students}(\text{sid}, \text{name}, \text{login}, \text{age}, \text{gpa})
  \]

- **Find**
  - the average of the ages of all the students

- **Solution**
  
  \[
  \text{SELECT AVG(age)} \\
  \text{FROM Students;}
  \]

- **Other Operations**
  - **SUM** (summation of all the values in a column)
  - **MIN** (least value)
  - **MAX** (highest value)
  - **COUNT** (the number of values), e.g.

  \[
  \text{SELECT COUNT(*)} \\
  \text{FROM Students;}
  \]

  - COUNTs the number of Students!

---

Ordering

- **Given**
  \[
  \text{Students}(\text{sid}, \text{name}, \text{login}, \text{age}, \text{gpa})
  \]

- **List**
  - the students in (ascending) alphabetical order of name

- **Solution**

  \[
  \text{SELECT *} \\
  \text{FROM Students} \\
  \text{ORDER BY name;}
  \]

  - and that for DESCending ORDER is

  \[
  \text{SELECT *} \\
  \text{FROM Students} \\
  \text{ORDER BY name DESC;}
  \]

  - Default is ASC
Grouping

- Given
  \[ \text{Students}(\text{sid, name, login, age, gpa}) \]

- Find
  - the names of students with gpa=4.0 and
  - group people with like ages together

- Solution

```sql
SELECT name
FROM Students
WHERE gpa=4.0
GROUP BY name;
```

More on Grouping

- Given
  \[ \text{Students}(\text{sid, name, login, age, gpa}) \]

- Find
  - the names of students with gpa=4.0 and
  - group people with like ages together and
  - show only those groups that have more than 2 students in it

- Solution

```sql
SELECT name
FROM Students
WHERE gpa=4.0
GROUP BY name
HAVING COUNT(*) > 2;
```
Summary of SQL Syntax

- General Form
  
  ```sql
  SELECT <attribute(s)>
  FROM <relation(s)>
  WHERE <condition(s)>
  GROUP BY <attribute(s)>
  HAVING <grouping condition(s)>
  ```

- Order of Execution
  
  - FROM
  - WHERE
  - GROUP BY
  - HAVING
  - SELECT

Views

- Can be viewed as temporary relations
  - do not exist physically BUT
  - can be queried and modified (sometimes) just like normal relations

- Example:

  ```sql
  CREATE VIEW GoodStudents(id,name) AS
  SELECT sid,name
  FROM Students
  WHERE gpa=4.0;

  SELECT *
  FROM GoodStudents
  WHERE name='Mark';

  Can we update the original relation using the GoodStudents VIEW?
  ```
Beginning of Wierd Stuff

- SQL uses Bag Semantics
  - meaning: does not normally eliminate duplicates
  - e.g. the SELECT clause

- BUT (a big BUT)

- this doesn’t apply to
  - UNION, INTERSECT and DIFFERENCE

- Either way, it provides facilities to do whatever we want

- If you want duplicates eliminated in SELECT clause
  - use SELECT DISTINCT.....

- If you want to prevent elimination of duplicates in UNION etc.
  - use (SELECT ...) UNION ALL (SELECT ...)
  - Likewise for INTERSECT and DIFFERENCE

... and that’s just the tip of the iceberg

- What happens with the following code?

```
SELECT R.A
FROM R, S, T
WHERE R.A = S.A or R.A = T.A
```

- when R(A) has {2,3}, S(A) has {3,4} and T(A) is {}

- Confusion Reigns!
Safety in Queries

☐ Some queries are inherently “unsafe”
☐ should not be permitted in DB access

☐ Example
☐ Given only the following relation

\[ \text{Students}(\text{id}) \]

☐ Find all those who are not students

☐ Easy to distinguish unsafe queries via common-sense
☐ Final result is not closed
☐ Is there an automatic way to determine “safety”?

Answer: Yes!

☐ Easiest to spot when written in Datalog

Answer(\text{id}) \leftarrow \text{NOTStudents}(\text{id}).

☐ Golden Rule
☐ Any variable that appears anywhere must also appear in a non-negated body part
☐ In this case, \text{id} causes the query to be unsafe

☐ Example of a Safe Query

Answer(\text{id}) \leftarrow \text{People}(\text{id}), \text{NOT Students}(\text{id})

☐ This produces all those people who are NOT students
☐ safe because the \text{People} relation provides a reference point
☐ \text{id} which appears in a negated body part also appears non-negated
More Dangers

- **Problem not restricted to negated body parts**
  - occurs even with arithmetic body parts (why?)

- **Given**
  - only the following relation

  \[ \text{Students}(id, age) \]

  - Find all those numbers that are greater than the age of some student

  \[ \text{Answer}(x) \leftarrow \text{Student}(id, age), x > age. \]

- **Extension to previous rule**
  - Any variable that appears anywhere must also appear in a non-negated, non-arithmetic body part
  - In this case, \( x \) causes the query to be unsafe
    - because it doesn’t appear in a non-negated, non-arithmetic part

One More Example

- **Given**
  - a relation \( \text{Composite}(x) \)
  - which lists all the composite numbers

- **Write a query to find**
  - the prime numbers

- **Wrong Way**
  - \( \text{Prime}(x) \leftarrow \text{NOT Composite}(x). \)

- **Right Way**
  - \( \text{Prime}(x) \leftarrow \text{Number}(x), \text{NOT Composite}(x). \)

- **Safety in Other Notations**
  - Relational Algebra: via the subtraction operator
  - SQL: via the EXCEPT construct

- **Notice how SQL and Relational Algebra do not allow unsafe queries**
  - because there is no way to write such queries with the given constructs
  - how clever, eh? :-
  - It is always amazing how “languages” force you to think in a certain manner
  - a problem long studied by philosophers
Recursion in Queries

- Used to specify an indefinite number of “applications” of a relation

- Example
  - Given only the following relation

    \[ \text{Person(name, parent)} \]

  - Find all the ancestors of “Mark”

- Easy to find an ancestor at a predefined level
  - parent: Use \text{Person}
  - grandparent: Join \text{Person} with \text{Person}
  - great-grandparent: Join \text{Person} with \text{Person} with \text{Person}
  - and so on.

- To find an ancestor at no predefined level
  - Need to join \text{Person} with \text{Person} an “indefinite” number of times

- SQL3 provides support for recursive definitions

Solution in Datalog

- First, the base case

  \[ \text{Ancestor(x,y)} \leftarrow \text{Person(x,y)}. \]

- Then, the inductive step

  \[ \text{Ancestor(x,y)} \leftarrow \text{Person(x,z)}, \text{Ancestor(z,y)}. \]

- Can also write the previous rule as

  \[ \text{Ancestor(x,y)} \leftarrow \text{Ancestor(x,z)}, \text{Ancestor(z,y)}. \]

  - why?
Recursion in SQL3

□ Use the “WITH RECURSIVE ... SELECT” construct

□ Example

\[
\text{WITH RECURSIVE Ancestor(name, ans) AS}
\]
\[
(\text{SELECT *}
\text{FROM Person})
\text{UNION}
\text{(SELECT Person.name, Ancestor.ans}
\text{FROM Person, Ancestor}
\text{WHERE Person.parent = Ancestor.name)}
\]
\[
\text{SELECT * FROM Ancestor;}
\]

□ Use with caution: Some kinds of recursive queries will not be allowed!
  □ example: the following Datalog query might not be allowed in SQL3
    \[
    \text{Ancestor}(x, y) \leftarrow \text{Ancestor}(x, z), \text{Ancestor}(z, y).
    \]
  □ because the rule involves 2 applications of the recursively defined predicate
  □ “Linear recursion” allows only one (as in the SQL code above)

Final Example

□ Be careful when combining negation, aggregation and recursion
  □ perfect recipe for disaster!

□ Mutual Recursion
  □ \text{Odd}(x) \leftarrow \text{Number}(x), \text{NOT Even}(x).
  □ \text{Even}(x) \leftarrow \text{Number}(x), \text{NOT Odd}(x).

□ What are the problems?
  □ Notice that the query appears “safe” (per Slide 96)
  □ cycles indefinitely!; no proper base cases

□ Illegal in SQL3
  □ not because of mutual recursion
  □ but due to the fact that there is no “unique interpretation” to the query
  □ Eg: 6 could be either in \textbf{Odd} or in \textbf{Even}; both are acceptable!

□ Sometimes mutual recursion is good and fruitful, if written properly
  □ with proper limiting constraints and base cases
Introduction to Deductive DBMSs

- Intersection of traditional RDBMSs and Logic Programming

- Example Systems
  - CORAL (Univ. Wisc.)
  - LDL++ (MCC)
  - XSB Systems (SUNY, Stony Brook)

- Can be viewed as
  - extending PROLOG-type systems with secondary storage
  - extending RDBMSs with deductive functionality

- Mappings: Commonalities between PROLOG and DBMSs
  - Predicate: Relation
  - Argument: Attribute
  - Ground Fact: Tuple
  - Extensional Definition: Table (defined by data)
  - Intensional Definition: Table (defined by a view)

PROLOG vs. RDBMSs

- Characteristics of PROLOG
  - Tuple-at-a-time
  - Backward Chaining
  - Top-Down
  - Goal-Oriented
  - Fixed-Evaluation Strategy (Depth-First)

- Characteristics of RDBMSs
  - Set-at-a-time (recall relational algebra)
  - Forward Chaining
  - Bottom-Up
  - Query Optimizer figures a good evaluation strategy

- Example
  - `ancestor(X,X). parent(amy,bob).`
  - `ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).`

- Query
  - Find the ancestors of bob: `ancestor(X,bob)`?
PROLOG Pitfalls

☐ Previous Example
  ☐ Linear Recursion
  ☐ Tail Recursion

☐ What if we reverse the order of clauses in
  ☐ ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).
  ☐ PROLOG goes into an infinite loop (why?)

☐ What if we make it
  ☐ ancestor(X,Y) <- ancestor(X,Z), ancestor(Z,Y).
  ☐ “Not Linear” Recursion

☐ Inference = Resolution + Unification
  ☐ Entailment in First Order Logic is Semi-decidable

Example of Deductive Query Optimization

☐ Same-Generation: Hello World of DDBMSs
  ☐ sg(X,Y) <- flat(X,Y).
  ☐ sg(X,Y) <- up(X,U), sg(U,V), down(V,Y).

☐ Magic: A Rewriting Technique
  ☐ Rewrite query such that advantages of bottom-up evaluation
  ☐ goal-oriented behavior are combined

☐ Example: For the query
  ☐ sg(john,Z)?

☐ Magic produces
  ☐ sg(X,Y) <- magic_sg(X), flat(X,Y).
  ☐ sg(X,Y) <- magic_sg(X), up(X,U), sg(U,V), down(V,Y).
  ☐ magic_sg(john).

☐ How do you know when to stop?
  ☐ Iterative Fixpoint Evaluation (when the answer stops changing)
SQL in a Programming Environment

- Incorporating SQL in a complete application

- Why?
  - There are some things we cannot do with SQL alone
    - e.g. preserving complex states, looping, branching etc.
  - Typically embed SQL in a host-language interface

- Problems: Impedance Mismatch
  - SQL operates on sets of tuples
  - Languages such as C, C++ operate on an individual basis

- Solution
  - easy when SELECT returns only one row

- When more than one row is returned
  - design an iterator to “run” over the results
  - called a “cursor”

How are these implemented?

- Vendor-Specific Implementations
  - ORACLE: PL/SQL (procedural extensions to SQL)

- Open Database Connectivity Standard
  - Provides a standard API for transparent database access
  - used when “database independence” is important
  - used when required to “connect” to diverse data sources
Tradeoffs

- **ODBC**
  - originated by Microsoft in 1991
  - adds one more abstraction layer
  - not as fast as a native API (does not exploit “special features”)
  - least-common denominator approach
  - constantly evolving

- **PL/SQL etc.**
  - “tailored” to the details of the underlying DBMS
  - might not extend to heterogeneous domains
  - modeled after a specific programming language (e.g. Ada for PL/SQL)

---

In Between: Stored Procedures

- **Used for developing “tightly-coupled” applications**
  - “push computations” selectively into the database system
  - avoid performance degradation
  - work in database address space instead of application address space

- **Advantages**
  - No sending SQL statements to and fro
  - eliminate pre-processing
  - speedup by an order of magnitude

- **Example Applications**
  - Database Administration
  - Integrity Maintenance and Checks
  - Database Mining

- **Disadvantages**
  - Non-standard implementation
  - Difficult to enforce transactional synchronization
  - Without traditional SQL optimization, can lead to performance degradation