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

CS 5614: Basic Data Definition and Modification in SQL

68

Example:

☐ Create a Relation/Table in 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

CS 5614: Basic Data Definition and Modification in SQL

70

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)
    ALTER TABLE Students DROP login;
    □ 'Add' an attribute (column)
    ALTER TABLE Students ADD phone CHAR(7);
    □ What happens to the new entry for the old records?
    □ Default is 'NULL' or say
    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)
```

CS 5614: Basic Data Definition and Modification in SQL

72

How do you enter/modify data?

```
☐ INSERT command
```

```
INSERT
INTO Students
VALUES (`53688','Mark','mark2345',23,3.9)

□ Cumbersome (use bulk loading; described later)

□ DELETE command

DELETE
FROM Students S
WHERE S.name = `Smith'

□ UPDATE command

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
    CREATE DOMAIN Email As CHAR(8) DEFAULT 'unknown';
    login Email //instead of login CHAR(8) DEFAULT 'unknown'
    □ Advantages: can be reused
    junkaddress Email,
    fromaddress Email,
    toaddress Email,
    ....
    □ Can DROP DOMAINS too!
    DROP DOMAIN Email;
    □ Affects only future declarations
```

CS 5614: Basic Data Definition and Modification in SQL

74

Keys

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

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

□ Whats 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

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! :-)
```

CS 5614: Basic Data Definition and Modification in SQL

76

Other Properties

```
□ 'NOT NULL' instead of DEFAULT

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
```

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) <- R(x,y,z,w). T(x,y,z,w) <- 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!

```
FROM R)
UNION
(SELECT *
FROM S)
```

- 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• $T(x,y,z,w) \leftarrow R(x,y,z,w)$, 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∩S
 T(x,y,z,w) <- 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.
 - $\pi_{name,address}R$
 - $T(x,y) \leftarrow R(x,y,z,w)$. 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'.

- $\sigma_{name="Michael"}R$
- $T(x,y,z,w) \leftarrow R(x,y,z,w), x='Michael'.$
- 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'.

$$T(x,y,z,w) \leftarrow R(x,y,z,w), z \leftrightarrow M', x \leftrightarrow Fox'.$$

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

$$T(x,y,z,w) \leftarrow R(x,y,z,w), z \leftrightarrow M'.$$

 $T(x,y,z,w) \leftarrow R(x,y,z,w), x \leftrightarrow Fox'.$

- 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:
 - \bullet $R \times S$
 - $T(x1,y1,z1,w1,x2,y2,z2,w2) \leftarrow R(x1,y1,z1,w1), S(x2,y2,z2,w2).$
 - SELECT R.name, R.address, R.gender, R.birthdate, S.name, S.address, S.gender, S.birthdate 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:
 - $R \bowtie_{R.name = `Michael'} S$
 - $T(x1,y1,z1,w1,x2,y2,z2,w2) \leftarrow R(x1,y1,z1,w1), S(x2,y2,z2,w2), x1='Michael'.$
 - SELECT 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:
 - \bullet $R\bowtie L$
 - $T(x,y,z,v,w) \leftarrow R(x,y,z,w), L(x,y,v).$
 - SELECT 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 m1, m2 and m3. This is most useful with relational algebra, like so:
 - $\bullet \ \rho_{M(m1,m2,m3)}(R)$

What is still to be co	vered
(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)

CS 5614: Misc. SQL Stuff, Safety in Queries

82

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

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
```

CS 5614: Misc. SQL Stuff, Safety in Queries

84

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
          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

```
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

CS 5614: Misc. SQL Stuff, Safety in Queries

86

SQL Subqueries

```
☐ Given
        Students(sid,name,login,age,gpa)
        HasCar(sid,carname)
□ Find
   □ the car of the student with login="mark"
□ Traditional Way
          SELECT carname
          FROM Students, HasCar
          WHERE Students.login='mark'
          AND Students.sid=HasCar.sid;
☐ The 'Subway'
          SELECT carname
          FROM HasCar
          WHERE sid=
              (SELECT sid FROM Students
               WHERE login='mark');
```

Aggregation

```
☐ Given

Students(sid,name,login,age,gpa)

☐ Find
☐ the average of the ages of all the students

☐ Solution

SELECT AVG(age)
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.

SELECT COUNT(*)
FROM Students;
```

 $\hfill\square$ COUNTs the number of Students!

CS 5614: Misc. SQL Stuff, Safety in Queries

□ Default is ASC

88

Ordering

```
☐ Given

Students(sid,name,login,age,gpa)
☐ List
☐ the students in (ascending) alphabetical order of name
☐ Solution

SELECT *
FROM Students
ORDER BY name;
☐ and that for DESCending ORDER is

SELECT *
FROM Students
ORDER BY name DESC;
```

Grouping

☐ Given
Students(sid,name,login,age,gpa)
☐ Find
☐ the names of students with gpa=4.0 and
☐ group people with like ages together
☐ Solution

SELECT name
FROM Students
WHERE gpa=4.0

GROUP BY name;

CS 5614: Misc. SQL Stuff, Safety in Queries

90

More on Grouping

☐ Given

Students(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

SELECT name
FROM Students
WHERE gpa=4.0
GROUP BY name
HAVING COUNT(*) > 2;

Summary of SQL Syntax

□ General Form

```
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

CS 5614: Misc. SQL Stuff, Safety in Queries

92

Views

☐ Can be viewed as temporary relations

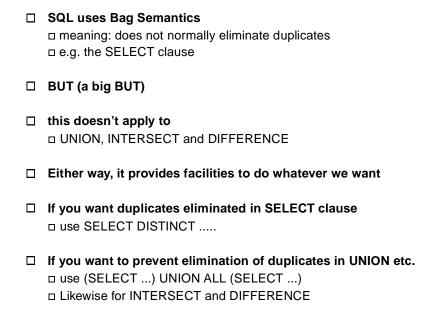
□ do not exist physically BUT

- □ can be queried and modified (sometimes) just like normal relations
- □ Example:

```
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



CS 5614: Misc. SQL Stuff, Safety in Queries

94

... 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

Students(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"?

CS 5614: Misc. SQL Stuff, Safety in Queries

96

Answer: Yes!

☐ Easiest to spot when written in Datalog

Answer(id) <- NOTStudents(id).</pre>

- ☐ Golden Rule
 - □ Any variable that appears anywhere must also appear in a non-negated body part
 - ☐ In this case, id causes the query to be unsafe
- □ Example of a Safe Query

Answer(id) <- People(id), NOT Students(id)</pre>

- ☐ This produces all those people who are NOT students
- □ safe because the **People** relation provides a reference point
- □ 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
 Students(id,age)
 □ Find all those numbers that are greater than the age of some student
 Answer(x) <- 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

bcoz it doesn't appear in a non-negated, non-arithmetic part

CS 5614: Misc. SQL Stuff, Safety in Queries

98

One More Example

□ a relation Composite(x) □ which lists all the composite numbers
Write a query to find □ the prime numbers
Wrong Way □ Prime(x) <- NOT Composite(x).
Right Way □ Prime(x) <- Number(x), 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
Person(name,parent)
□ Find all the ancestors of "Mark"
Easy to find an ancestor at a predefined level □ parent: Use Person □ grandparent: Join Person with Person □ great-grandparent: Join Person with Person with Person □ and so on.
To find an ancestor at no predefined level □ Need to join Person with Person an "indefinite" number of times
SQL3 provides support for recursive definitions

CS 5614: Misc. SQL Stuff, Safety in Queries

100

Solution in Datalog

```
□ First, the base case
Ancestor(x,y) <- Person(x,y).
□ Then, the inductive step
Ancestor(x,y) <- Person(x,z), Ancestor(z,y).
□ Can also write the previous rule as
Ancestor(x,y) <- Ancestor(x,z), Ancestor(z,y).
□ why?</pre>
```

Recursion in SQL3

```
□ Use the "WITH RECURSIVE ... SELECT" construct
□ Example

WITH RECURSIVE Ancestor(name, ans) AS

(SELECT *
FROM Person)
UNION
(SELECT Person.name, Ancestor.ans
FROM Person, Ancestor
WHERE Person.parent=Ancestor.name)

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
Ancestor(x,y) <- Ancestor(x,z), 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)
```

CS 5614: Misc. SQL Stuff, Safety in Queries

102

Final Example

Be careful when combining negation, aggregation and recursion perfect recipe for disaster!
<pre>Mutual Recursion □ Odd(x) <- Number(x), NOT Even(x). □ Even(x) <- Number(x), NOT Odd(x).</pre>
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 Odd or in Even; both are acceptable!
Sometimes mutual recursion is good and fruitful, if written properly up 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)

CS 5614: Misc. SQL Stuff, Safety in Queries

104

PROLOG vs. RDBMSs

	Characteristics of PROLOG
	□ Tuple-at-a-time
	□ Backward Chaining
	□ Top-Down
	□ Goal-Oriented
	□ Fixed-Evaluation Strategy (Depth-First)
П	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).
	\square ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).
П	Query
	□ Find the ancestors of bob: ancestor(X,bob)?
	ind the ancestors of bob. ancestor (A, 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

CS 5614: Misc. SQL Stuff, Safety in Queries

106

Example of Deductive Query Optimization

SQL in a Programming Environment

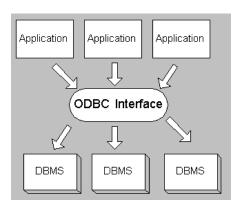
- ☐ Incorporating SQL in a complete application
- □ Why?
 - ☐ There are some things we cannot do with SQL alone ☐ e.g. preserving complex states, looping, branching etc.
 - □ Typically embed SQL in a host-language interface
- □ Problems: Impedance Mismatch
 - □ SQL operates on sets of tuples
 - □ Languages 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"

CS 5614: Misc. SQL Stuff, Safety in Queries

108

How are these implemented?

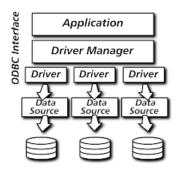
- □ Vendor-Specific Implementations
 - □ ORACLE: PL/SQL (procedural extensions to SQL)
- □ Open Database Connectivity Standard
 - □ Provides a standard API for transparent database access
 - □ used when "database independence" is important
 - □ used when required to "connect" to diverse data sources



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 PI/SQL)

CS 5614: Misc. SQL Stuff, Safety in Queries

110

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 Adminstration
- □ Integrity Maintenance and Checks
- □ Database Mining

□ Disadvantages

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



Introduction to Query Optimization

- ☐ Helps attain declarativeness of RDBMSs
 - □ One of the main reasons for commercial success of DBMSs
- □ A motivating example
 - ☐ Find all students with 4.0 gpa enrolled in CS5614

SELECT name
FROM Students, Classroll
WHERE Students.name = Classroll.studentname
AND Students.gpa = 4.0
AND Classroll.coursename = `CS5614'

- ☐ Two Strategies
 - □ Do join and then filter out the ones with gpa <> 4.0 and course <> CS5614 □ Filter first the ones with gpa <> 4.0 and course <> CS5614 and then Join
- □ Which is Better?
 - □ Always good to "push selections" as far down into the query parse tree

CS 5614: Misc. SQL Stuff, Safety in Queries

112

How does a Query Optimizer work?

- ☐ Three Requirements
 - □ A Search Space of "Plans"
 - □ A Cost Model (for Plan evaluation)
 - □ An Enumeration Algorithm
- □ Ideally
 - ☐ Search Space: contains both good and efficient plans
 - □ Cost Models: cheap to compute and accurate
 - □ Enumeration Algorithm: efficient (not a monkey-typewriter algorithm)
- □ Example of a Search Space
 - □ See Previous Slide
- □ Examples of Cost Models
 - □ #(tuples) evaluation
 - □ #(main memory locations) etc.
- □ Example of an enumeration algorithm
 - □ Sequential enumeration of a lattice of plans
 - □ Dynamic Programming vs. Greedy Approaches

A Simple Measure of "Cost"

☐ #(Tuples) in a query

□ Easiest to compute for

□ Cartesian Product: #(R X S) = #(R)#(S)

 \square Projection:#(Pi(R)) = #(R)

□ A Notation for Other Operations

 \square V(R,A) = Number of distinct values of attribute "A" in R

□ formulas assume that all values of "A" are equally likely in R

□ Holds in average case for most distributions (e.g. Zipf)

□ Selectivity Factors for Selection Operations

□ Equality Tests: Use 1/V(R,A)

 \square < or > Tests: Use 1/3

□ "<>" Test: Use (V(R,A)-1)/V(R,A)

☐ AND Conditions: Multiply Selectivity Factors

□ OR Conditions: Three Choices

□ Sum of results from individual selectivity factors

□ Max(sum,total size of relation): why?

 \square n(1-(1-m1/n)(1-m2/n)) formula : most accurate

CS 5614: Misc. SQL Stuff, Safety in Queries

114

Estimating the Size of a Join

 \square Assume: R(X,Y) Join S(Y,Z)

□ Range of Values

□ Minimum: 0

□ In-between: #(R) (if Y is a foreign key for R and a key for S)

□ Maximum: #(R)#(S) (if Y's in R and S are all the same)

□ Assumptions for Join Size Estimation

□ Containment of Value Sets

□ Preservation of Value Sets

□ Containment of Value Sets

 \square If $V(R,Y) \leftarrow V(S,Y)$ then the Y's in R are a subset of the Y's in S

□ Satisfied when Y is a foreign key in R and a key in S

□ Preservation of Value Sets

 \square #(R Join S,X) = #(R,X)

 \square #(R Join S,Z) = #(S,Z)

□ why is this reasonable?

The Actual Estimate

 \square Assume that $V(R,Y) \leftarrow V(S,Y)$ □ Every tuple in R has a chance of 1/V(S,Y) of joining with a tuple of S □ Every tuple in R has a chance of #(S)/V(S,Y) of joining with S □ All tuples in R have a chance of #(R)#(S)/V(S,Y) of joining with S \square What if $V(S,Y) \leftarrow V(R,Y)$ \square Answer: #(R)#(S)/V(R,Y) \square In general: $\#(R)\#(S)/(\max(V(S,Y),V(R,Y)))$ ☐ What if there are multiple join attributes □ Have a "max" factor in the denominator for each such attribute! ☐ How to Estimate #(R Join S Join T)? □ Does it matter which we do first? □ Surprise! □ Estimation formula preserves associativity of Joins! □ In other words, "it takes care of itself!" ☐ Thus, for a Join attribute appearing > 2 times ☐ 3 times: Use two highest values ☐ 4 times: Use three highest values etc.

CS 5614: Misc. SQL Stuff, Safety in Queries

116

More on Join Associativity and Commutativity

□ Which is better: (R Join S) or (S Join R)
□ Good to put the smaller relation on the left
□ Why? Most Join algorithms are assymmetric
□ Example:
□ Construct a "good query tree" for the following

SELECT movietitle
FROM Actors, ActedIn
WHERE Actors.name = ActedIn.actorname

AND Actors.age = 23

- ☐ Number of Possible Trees of n Attributes
 - ☐ Arises from the shape of the trees: T(n)
 - ☐ Arises from permuting the leaves: n!
 - □ Total choices: n! T(n)

What is T(n)?

```
□ Sample Values
```

- □ 1: 1
- □ 2: 1
- □ 3: 2
- □ 4: 5
- □ 5: 14

□ A formula

```
□ T(1) = 1 (Basis)
```

$$\Box T(n) = T(1)T(n-1) + T(2)T(n-2) + + T(n-1)T(1)$$

□ Classifications

- □ Left-Deep Trees: All right children are leaves
- □ Right-Deep Trees: All left children are leaves
- ☐ Bushy Trees: Neither Left-Deep nor Right-Deep

☐ Choosing a Join Order: Restricted to Left-Deep Trees

- □ By Dynamic Programming: O(n!)
- ☐ Greedy Approach: Make local selections

CS 5614: Misc. SQL Stuff, Safety in Queries

118

Example

```
□ Consider
```

```
\square R(a,b): #(R) = 1000, V(R,a) = 100, V(R,b) = 200
```

- \Box S(b,c): #(S) = 1000, V(S,b) = 100, V(S,c) = 500
- \Box T(c,d): #(T) = 1000, V(T,c) = 20, V(T,d) = 50

□ Possible Join Orders

- □ (R Join S) Join T
- ☐ (S Join R) Join T (same as above; why?)
- □ (R Join T) Join S
- ☐ (T Join R) Join S (same as above)
- □ (S Join T) Join R
- □ (T Join S) Join R (same as above)

☐ Cost Estimation = Sizes of Intermediate Relations

- □ (R Join S) Join T: 5000
- ☐ (R Join T) Join S: 1000000
- □ (S Join T) Join R: 2000
- ☐ Best Plan = (S Join T) Join R

Database Tuning: Why?

□ Two Families of Queries

- □ OLTP (Access small number of records)
- □ OLAP (Summarize from a large number of records)

□ Sources of Poor Performance

- □ Imprecise Data Searches
- □ Random vs. Sequential Disk Accesses
- □ Short Bursts of Database Interaction
- □ Delays due to Multiple Transactions

□ What can be done?

- □ Tune Hardware Architecture
- □ Tune OS
- □ Tune Data Structures and Indices

CS 5614: Misc. SQL Stuff, Safety in Queries

120

Examples of Database Tuning

□ To normalize or not to

- □ Sacrificing Redundancy Elimination
- □ Sacrificing Dependency Elimination

$\ \square$ Several Choices of Normalized Schemas

□ Vertical Partitioning Applications

□ Recomputing Indices

□ Histograms etc. might be outdated

□ Restricting Uses of Subqueries

□ Unnesting query blocks by Joins

□ Declining the Use of Indices

- □ Table Scans for Small Tables
- □ Rule-based optimization: Rewrite A=6 as "A+0=6"

□ Provide Redundant Tables

□ Decision-Support/ Data Mining Queries