The SQL Query Language

- Developed by IBM (system R) in the 1970s
- Need for a standard since it is used by many vendors
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision)
  - SQL-99 (major extensions)
  - SQL 2003 (XML ↔ SQL)

Creating Relations in SQL

- **CREATE TABLE Acct**
  (bname CHAR(20),
  acctn CHAR(20),
  bal REAL,
  PRIMARY KEY (bname, acctn),
  FOREIGN KEY (bname REFERENCES branch)
- **CREATE TABLE Branch**
  (bname CHAR(20),
  bcity CHAR(30),
  assets REAL,
  PRIMARY KEY (bname))
- Observe that the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.

Referential Integrity in SQL

- SQL-92 on support all 4 options on deletes and updates.
  - Default is NO ACTION (*delete/update is rejected*)
  - CASCADE (also delete all tuples that refer to deleted tuple)
  - SET NULL / SET DEFAULT (sets foreign key value of referencing tuple)

CREATE TABLE Acct
(bname CHAR(20) DEFAULT 'main',
acctn CHAR(20),
bal REAL,
PRIMARY KEY (acctn), *note change from prev*
FOREIGN KEY (bname) REFERENCES Branch
ON DELETE SET DEFAULT)

Primary and Candidate Keys in SQL

- **Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.**
- There at most one book with a given title and edition – date, publisher and isbn are determined
- Used carelessly, an IC can prevent the storage of database instances that arise in practice! Title and ed suffice? UNIQUE (title, ed, pub)?

CREATE TABLE Book
(isbn CHAR(10)
title CHAR(100),
ed INTEGER,
pub CHAR(30),
date INTEGER,
PRIMARY KEY (isbn),
UNIQUE (title, ed))

Destroying and Alterating Relations

DROP TABLE Acct

Destroys the relation Acct. The schema information the tuples are deleted.

ALTER TABLE Acct
ADD COLUMN Type CHAR (3)

Adds a new field; every tuple in the current instance is extended with a *null* value in the new field.

Adding and Deleting Tuples

Basics:

- To insert a single tuple:
  INSERT INTO Branch (bname, bcity, assets)
  VALUES ('Nassau ST. ', 'Princeton', 7320571.00

  (bname, bcity, assets) *optional*

- To delete all tuples satisfying some condition:
  DELETE FROM Acct A
  WHERE A.acctn = 'B7730'

- To update:
  UPDATE Branch B
  SET B.bname = 'Nassau East'
  WHERE B.bname = 'Nassau St.'

Basic SQL Query

SELECT	[DISTINCT] *select-list*
FROM	*from-list*
WHERE	*qualification*

- ***from-list*** **A list of relation names (possibly with a *range-variable* after each name).**
- ***select-list*** **A list of attributes of relations in *from-list***
- ***qualification*** **Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of <, >,=, ≤, ≥, ≠) combined using AND, OR and NOT.**
- **DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!**

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *from-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *select-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example Instances

- We will use these instances of the Acct and Branch relations in our examples.

R1

bname	bcity	assets
pu	Pton	10
nyu	nyc	20
time sq	nyc	30

S1

bname	acctn	bal
pu	33	356
nyu	45	500

Example of Conceptual Evaluation

```
SELECT R.acctn
FROM   Branch S, Acct R
WHERE  S.bname=R.bname AND S.assets<20
```

bname	bcity	assets	bname	acctn	bal
pu	**Pton**	**10**	**pu**	**33**	**356**
pu	Pton	**10**	nyu	45	500
nyu	nyc	20	pu	33	356
nyu	nyc	20	**nyu**	45	500
time sq	nyc	30	pu	33	356
time sq	nyc	30	nyu	45	500

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT R.acctn
FROM   Branch S, Acct R
WHERE  S.bname=R.bname
       AND assets<20
```

OR

```
SELECT acctn
FROM   Branch, Acct
WHERE  Branch.bname=Acct.bname AND assets<20
```

It is good style, however, to use range variables always!

Find branches with at least one acct and their cities

```
SELECT S.bname, S.bcity
FROM Branch S, Acct R
WHERE S.bname=R.bname
```

- Would adding DISTINCT to this query make a difference?
- What if only SELECT S.bcity ? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

```
SELECT A.name, age=2003-A.yrofbirth
FROM Alumni A
WHERE A.dept LIKE 'C%S'
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find pairs (Alumnus(a) name and age defined by year of birth) for alums whose dept. begins with "C" and ends with "S".*
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters.


```
CREATE TABLE Acct
   (bname CHAR(20),
    acctn CHAR(20),
    bal REAL,
    PRIMARY KEY ( acctn),
    FOREIGN KEY (bname REFERENCES Branch )
```

note is 2nd version

```
CREATE TABLE Branch
    (bname CHAR(20),
     bcity CHAR(30),
     assets REAL,
     PRIMARY KEY (bname) )
```

```
CREATE TABLE Cust
    (name CHAR(20),
     street CHAR(30),
     city CHAR(30),
     PRIMARY KEY (name) )
```

```
CREATE TABLE Owner
  (name CHAR(20),
   acctn CHAR(20),
   FOREIGN KEY (name REFERENCES Cust )
   FOREIGN KEY (acctn REFERENCES Acct ) )
```


Find names of customers with accts in branches in Princeton or West Windsor (WW)

- **UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).**
- **If we replace OR by AND in the first version, what do we get?**
- **Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)**

```
SELECT D.name
FROM Acct A, Owner D, Branch B
WHERE D.acctn=A.acctn AND
  A.bname=B.bname AND (B.bcity=
  'Princeton' OR B.bcity='WW')
```

```
SELECT D.name
FROM Acct A, Owner D, Branch B
WHERE D.acctn=A.acctn AND
  A.bname=B.bname AND B.bcity=
  'Princeton'
UNION
SELECT D.name
FROM Acct A, Owner D, Branch B
WHERE D.acctn=A.acctn AND
  A.bname=B.bname AND B.bcity='WW'
```


Find names of customers with accts in branches in Princeton and West Windsor (WW)

INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.

Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```
SELECT C.name
FROM Cust C, Acct A1, Acct A2, Owner D1,
  Owner D2, Branch B1, Branch B2
WHERE C.name=D1.name AND
  C.name=D2.name AND
  D1.acctn=A1.acctn AND D2.acctn=A2.acctn AND
  A1.bname=B1.bname AND A2.bname=B2.bname
  AND B1.bcity='Princeton' AND B2.bcity='WW'
```

Refers to Key field!

```
SELECT D.name
FROM Acct A, Owner D, Branch B
WHERE D.acctn=A.acctn AND
  A.bname=B.bname AND B.bcity=
  'Princeton'
INTERSECT
SELECT D.name
FROM Acct A, Owner D, Branch B
WHERE D.acctn=A.acctn AND
  A.bname=B.bname AND B.bcity='WW'
```

Find names of customers with accts in branches in Princeton and West Windsor (WW)

```
SELECT D1.name
FROM  Acct A1, Acct A2, Owner D1, Owner D2, Branch B1, Branch B2
WHERE D1.name=D2.name AND D1.acctn=A1.acctn AND
  D2.acctn=A2.acctn AND A1.bname=B1.bname AND
  A2.bname=B2.bname AND B1.bcity='Princeton' AND B2.bcity='WW'
```

Refers to key field Cust.name!

```
SELECT D.name
FROM Acct A, Owner D, Branch B
WHERE D.acctn=A.acctn AND
  A.bname=B.bname AND B.bcity= 'Princeton'
INTERSECT
SELECT D.name
FROM Acct A, Owner D, Branch B
WHERE D.acctn=A.acctn AND
  A.bname=B.bname AND B.bcity='WW'
```


Nested Queries

Find names of all branches with accts of cust. who live in Rome

```
SELECT A.bname
FROM Acct A
WHERE A.acctn IN (SELECT D.acctn
                  FROM Owner D, Cust C
                  WHERE D.name = C.name AND C.city='Rome')
```

A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)

What get if use NOT IN?

To understand semantics of nested queries, think of a nested loops evaluation: *For each Acct tuple, check the qualification by computing the subquery.*

Nested Queries with Correlation

Find acct no.s whose owners own at least one acct with a balance over 1000

```
SELECT D.acctn
FROM Owner D
WHERE EXISTS (SELECT *
              FROM Owner E, Acct R
              WHERE R.bal>1000 AND R.acctn=E.acctn
                    AND E.name=D.name)
```

- **EXISTS is another set comparison operator, like IN.**
- **If UNIQUE is used, and * is replaced by *E.name*, finds acct no.s whose owners own no more than one acct with a balance over 1000. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *E.name*?)**
- **Illustrates why, in general, subquery must be re-computed for each Branch tuple.**

More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL, *op* in $>,<,=,\geq,\leq,\neq$
- *Find names of branches with assests at least as large as the assets of some NYC branch:*

```
SELECT B.bname
FROM Branch B
WHERE B.assets ≥ ANY (SELECT Q.assets
                      FROM Branch Q
                      WHERE Q.bcity='NYC')
```

Includes NYC branches?

Division in SQL

Find tournament winners who have won all tournaments.

```
SELECT  R.wname
FROM Winners R
WHERE  NOT EXISTS
          ((SELECT  S.tourn
            FROM  Winners S)
           EXCEPT
            (SELECT  T.tourn
             FROM  Winners T
             WHERE  T.wname=R.wname))
```

```
CREATE TABLE Winners
   (wname  CHAR((30),
    tourn  CHAR(30),
    year   INTEGER)
```


Division in SQL – template

Find name of all customers who have accounts at all branches in Princeton.

```
SELECT
FROM
WHERE  NOT EXISTS
          ((SELECT
               FROM
                WHERE                              )
            EXCEPT
           (SELECT
                FROM
                WHERE                              )
```


Division in SQL – our example

Find name of all customers who have accounts at all branches in Princeton.

```
SELECT  C.name
FROM  Cust C
WHERE  NOT EXISTS
          ((SELECT  B.bname
            FROM  Branch B
             WHERE B.bcity = 'Princeton')
           EXCEPT
            (SELECT  A.bname
             FROM  Acct A, Owner D
             WHERE  A.acctn = D.acctn
                        AND D.name = C.name))
```


Aggregate Operators

- Significant extension of relational algebra.

```
COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)
```

single column

```
SELECT  COUNT (*)
FROM  Acct R
```

```
SELECT  AVG ( DISTINCT R.bal)
FROM  Acct R
WHERE  R.bname='nyu'
```

```
SELECT  AVG (R.bal)
FROM  Acct R
WHERE  R.bname='nyu'
```

```
SELECT  S.bname
FROM  Branch S
WHERE  S.assets=
     (SELECT  MAX(T.assets)
          FROM  Branch T)
```

```
SELECT  COUNT (DISTINCT S.bcity)
FROM  Branch S
```

Find name and city of the poorest branch

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- Is it poorest *branch* or poorest *branches*?

```
SELECT S.bname, MIN (S.assets)
FROM Branch S
```

```
SELECT S.bname, S.assets
FROM Branch S
WHERE S.assets =
        (SELECT MIN (T.assets)
         FROM Branch T)
```


GROUP BY and HAVING

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the maximum assets of all branches in a city for each city containing a branch.*
  - If we know all the cities we could write a query for each city:

```
SELECT MAX(B.assests)
FROM Branch B
WHERE B.bcity='nyc'
```

  - Not elegant. Worse: what if add or delete a city?

Queries With GROUP BY and HAVING

```
SELECT     [DISTINCT] select-list
FROM       from-list
WHERE      qualification
GROUP BY   grouping-list
HAVING     group-qualification
```

- The *select-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

- The cross-product of *from-list* is computed, tuples that fail *qualification* are discarded, \`*unnecessary'* fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group*!
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

What fields are unnecessary?

↓

What fields are necessary:

Exactly those mentioned in
SELECT, GROUP BY or HAVING clauses

Find the maximum assets of all branches in a city for each city containing a branch.

SELECT B.bcity, MAX(B.assets)
FROM Branch B
GROUP BY B.bcity

empty WHERE and HAVING

bname	bcity	assets
pu	Pton	10
pmc	Pton	8
nyu	nyc	20
time sq	nyc	30

bcity	assets
Pton	10
Pton	8
nyc	20
nyc	30

bcity	
Pton	10
nyc	30

2nd column of result
is unnamed.
(Use AS to name it.)

For each city, find the average assets of all branches in the city that have assets under 25

SELECT B.bcity, AVG(B.assets) AS avg_assets
FROM Branch B
GROUP BY B.bcity
HAVING B.assets < 25

WRONG! Why?

For each city, find the average assets of all branches in the city that have assets under 25

SELECT B.bcity, AVG(B.assets) AS avg_assets
FROM Branch B
WHERE B.assets < 25
GROUP BY B.bcity

bcity	assets
Pton	10
Pton	8
nyc	20

bcity	avg_assets
Pton	9
nyc	20

*For each customer living in nyc (identified by **name**), find the total balance of all accounts in the bank*

```
SELECT C.name, SUM (A.bal) AS total
FROM Cust C, Owner D, Acct A
WHERE C.name=D.name AND D.acctn=A.acctn
GROUP BY C.name, C.city
HAVING C.city='nyc'
```

- Grouping over a join of three relations.
- Why are both C.name and C.city in GROUP BY?
  - Recall Cust.name is primary key
- What if we remove HAVING C.city='nyc' and add AND C.city='nyc' to WHERE

*For each cust. (id. by **name**) with an acct. in a NYC branch, find the total balance of all accts in the bank*

?

*For each cust. (id. by **name**) with an acct. in a NYC branch, find the total balance of all accts in the bank*

```
SELECT C.name, SUM (A2.bal) AS total
FROM Cust C, Owner D1, Owner D2, Acct A1, Acct A2,
     Branch B
WHERE C.name=D1.name AND C.name=D2.name AND
      D1.acctn=A1.acctn AND D2.acctn=A2.acctn AND
      A1.bname=B.bname AND B.bcity='nyc'
GROUP BY C.name
```

Why not

```
FROM Cust C, Owner D2, Acct A2, Branch B
WHERE C.name=D2.name AND D2.acctn=A2.acctn
      ANDA2.bname=B.bname AND B.bcity='nyc'
```


Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
  - SQL provides a special value *null* for such situations.
- The presence of *null* complicates many issues. E.g.:
  - Special operators needed to check if value is/is not *null.*
  - Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and *unknown*).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.

Joins in SQL

- SQL has both inner joins and *outer* join
- Use where need relation, e.g. "FROM …"
- Inner join variations as for relational algebra
  Cust INNER JOIN Owner ON
    Cust.name =Owner.name
  Cust INNER JOIN Owner USING (name)
  Cust NATURAL INNER JOIN Owner
- Outer join includes tuples that don't match
  - fill in with nulls
  - 3 varieties: left, right, full

Outer Joins

- *Left outer join of S and R:*
  - take inner join of S and R (with whatever qualification)
  - add tuples of S that are not matched in inner join, filling in attributes coming from R with "null"
- *Right outer join:*
  - as for left, but fill in tuple of R
- *Full outer join:*
  - both left and right

Example

Given Tables:

sid	college
77	Forbes
35	Mathey
21	Butler

sid	dept
77	ELE
21	COS
42	MOL

NATURAL INNER JOIN:

77	Forbes	ELE
21	Butler	COS

NATURAL LEFT OUTER JOIN add:

35	Mathey	null

NATURAL RIGHT OUTER JOIN add:

42	null	MOL

NATURAL FULL OUTER JOIN add ***both***

Views

- A *view* is just a relation, but we store a *definition*, rather than a set of tuples.

```
CREATE VIEW
        YoungStudentGrades (name, grade)
        AS SELECT S.name, E.grade
        FROM Students S, Enrolled E
        WHERE S.sid = E.sid and S.age<21
```

- Views can be dropped using the DROP VIEW command.
  - How to handle DROP TABLE if there's a view on the table?
    - DROP TABLE command has options to let user specify this.

Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- *Types of IC's*: Domain constraints, primary key constraints, candidate key constraints, foreign key constraints, general constraints.

General Constraints

- Useful when more general ICs than keys are involved.

```
CREATE TABLE  GasStation
   ( name CHAR(30),
     street  CHAR(40),
     city CHAR(30),
     st CHAR(2),
     type CHAR(4),
     PRIMARY KEY  (name, street, city, st),
     CHECK  ( type='full' OR type='self' ),
     CHECK (st <>'nj' OR type='full')  )
```


More General Constraints

- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE  FroshSemEnroll
   ( sid  CHAR(10),
     sem_title CHAR(40),
     PRIMARY KEY  (sid, sem_title),
     FOREIGN KEY (sid) REFERENCES Students
     CONSTRAINT  froshonly
     CHECK  (2008 IN
               ( SELECT  S.classyear
                 FROM  Students S
                 WHERE  S.sid=sid) ) )
```


Constraints Over Multiple Relations

Number of bank branches in a city is less than 3 or the population of the city is greater than 100,000

- Cannot impose as CHECK on each table. If either table is empty, the CHECK is satisfied
- Is conceptually wrong to associate with individual tables
- ASSERTION is the right solution; not associated with either table.

Number of bank branches in a city is less than 3 or the population of the city is greater than 100,000

```
CREATE ASSERTION branchLimit
CHECK
( NOT EXISTS ( (SELECT C.name, C.state
                FROM Cities C
                WHERE C.pop <=100000 )
               INTERSECT
               ( SELECT D.name, D.state
                 FROM Cities D
                 WHERE 3 <=
                       (SELECT COUNT (*)
                        FROM Branches B
                        WHERE B.bcity=D.name ) ) ) )
```


Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)

Summary

- SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- Relationally complete; in fact, significantly more expressive power than relational algebra.
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.

Summary (Contd.)

- Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.
- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database

ERATTA

Old version slide #2 next page:

had ':' after column name

Other slides originally with same problem: 3, 5, 14

Creating Relations in SQL

- **CREATE TABLE Acct**
  (bname: CHAR(20),
  acctn: CHAR(20),
  bal: REAL,
  PRIMARY KEY (bname, acctn),
  FOREIGN KEY (bname REFERENCES branch)
- **CREATE TABLE Branch**
  (bname: CHAR(20),
  bcity: CHAR(30),
  assets: REAL,
  PRIMARY KEY (bname))
- Observe that the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.