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, current standard)
  - SQL-99 (major extensions)

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.

Destroying and Altering 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

- 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 $<, >, =, \leq, \geq, \neq$) 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

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Example of Conceptual Evaluation

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND R.bid=103
```

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

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 S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND bid=103
```

OR

```
SELECT sname
FROM   Sailors, Reserves
WHERE  Sailors.sid=Reserves.sid
       AND bid=103
```

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

Find sailors who've reserved at least one boat

```
SELECT S.sid
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid
```

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

```
SELECT A.name, age=2003-A.dob
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 date of birth) for alums whose dept. begins with "C" and contains "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),          note different than last time
    FOREIGN KEY (bname REFERENCES Branch )

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 Depos
   (name: CHAR(20),
    acctn: CHAR(20),
    FOREIGN KEY (name REFERENCES Cust )
    FOREIGN KEY (acctn REFERENCES Acct ) )
```

```
CREATE TABLE Sailors
    (sid: INTEGER,
     sname: STRING,
     rating: INTEGER,
     age: REAL,
     PRIMARY KEY ( sid)  )

CREATE TABLE Boats
     (bid: INTEGER,
      bname: STRING,
      color: STRING ,
      PRIMARY KEY (bid)  )

CREATE TABLE Reserves
    (sid: INTEGER,
     bid: INTEGER,
     day: DATE,
     FOREIGN KEY (sid) REFERENCES  Sailors,
     FOREIGN KEY (bid) REFERENCES  Boats    )
```


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, Depos 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, Depos D, Branch B
WHERE D.acctn=A.acctn AND
   A.bname=B.bname AND B.bcity=
   'Princeton'
UNION
SELECT  D.name
FROM Acct A, Depos 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, Depos D1,
  Depos 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, Depos D, Branch B
WHERE D.acctn=A.acctn AND
   A.bname=B.bname AND B.bcity=
   'Princeton'
INTERSECT
SELECT  D.name
FROM Acct A, Depos 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  Depos 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 names of sailors who've reserved boat #103:

```
SELECT  S.sname
FROM  Sailors S
WHERE  EXISTS  (SELECT  *
                FROM  Reserves R
                WHERE  R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN.
- If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)
- Illustrates why, in general, subquery must be re-computed for each Sailors 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 sailors who've reserved all boats.

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS
        ((SELECT B.bid
          FROM Boats B)
         EXCEPT
         (SELECT R.bid
          FROM Reserves R
          WHERE R.sid=S.sid))
```


Division in SQL – our example

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 Branches B
          WHERE B.bcity = 'Princeton')
         EXCEPT
         (SELECT A.bname
          FROM Acct A, Depos 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 Sailors S

SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10

SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
                 FROM Sailors S2)

SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'

SELECT AVG ( DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```


Find name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT S.sname, MAX (S.age)
FROM Sailors S

SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
        (SELECT MAX (S2.age)
         FROM Sailors S2)

SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
       FROM Sailors S2)
       = S.age
```


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 age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

For i = 1, 2, ... , 10:

```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```

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.

Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors

```
SELECT  S.rating, MIN (S.age)
FROM    Sailors S
WHERE   S.age >= 18
GROUP BY S.rating
HAVING  COUNT (*) > 1
```

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
71	zorba	10	16.0
64	horatio	7	35.0
29	brutus	1	33.0
58	rusty	10	35.0

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes `*unnecessary*'.
- 2nd column of result is unnamed. (Use AS to name it.)

rating	age
1	33.0
7	45.0
7	35.0
8	55.5
10	35.0

rating	
7	35.0

Answer relation

For each red boat, find the number of reservations for this boat

```
SELECT  B.bid, COUNT (*) AS scount
FROM    Sailors S, Boats B, Reserves R
WHERE   S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY B.bid
```

- Grouping over a join of three relations.
- What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?

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

```
Sailors INNER JOIN Reserves ON
  Sailors.sid =Reserved.sid
Sailors INNER JOIN Reserves USING (sid)
Sailors NATURAL INNER JOIN Reserves
```

- 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 YoungActiveStudents (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 the 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, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.

General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE Sailors
    ( sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK ( rating >= 1
            AND rating <= 10 )
CREATE TABLE Reserves
    ( sname CHAR(10),
    bid INTEGER,
    day DATE,
    PRIMARY KEY (bid,day),
    CONSTRAINT noInterlakeRes
    CHECK (`Interlake' <>
        ( SELECT B.bname
        FROM Boats B
        WHERE B.bid=bid)))
```


Constraints Over Multiple Relations

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

```
CREATE TABLE Sailors
    ( sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK
    ( (SELECT COUNT (S.sid) FROM Sailors S)
    + (SELECT COUNT (B.bid) FROM Boats B) < 100
```

Number of boats plus number of sailors is < 100

```
CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100
```

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)

Triggers: Example (SQL:1999)

```
CREATE TRIGGER youngSailorUpdate
    AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
    INSERT
        INTO YoungSailors(sid, name, age, rating)
        SELECT sid, name, age, rating
        FROM NewSailors N
        WHERE N.age <= 18
```


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.
- 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.