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)
- 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

- 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 of Conceptual Evaluation

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

We will use these instances of the Sailors and Reserves relations in our examples.

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?

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

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

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

Expressions and Strings

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

LIKE is used for string matching. '%' stands for any one character and '_' stands for 0 or more arbitrary characters.
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)

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'
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'

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)

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.
**Division in SQL**

Find sailors who’ve reserved all boats.

```sql
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)
```

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

```sql
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)
```

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.

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

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

**Aggregate Operators**

- Significant extension of relational algebra.

```sql
SELECT COUNT (*)
FROM Sailors S
```

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

**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 (!):

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

For \( i = 1, 2, \ldots, 10 \):
Queries With GROUP BY and HAVING

- 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.)

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

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
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>71</td>
<td>xorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

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) possibly 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

<table>
<thead>
<tr>
<th>sid</th>
<th>college</th>
<th>sid</th>
<th>dept</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>Forbes</td>
<td>77</td>
<td>ELE</td>
</tr>
<tr>
<td>35</td>
<td>Mathey</td>
<td>21</td>
<td>COS</td>
</tr>
<tr>
<td>21</td>
<td>Butler</td>
<td>42</td>
<td>MOL</td>
</tr>
</tbody>
</table>

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., name 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 ( (SELECT COUNT (S.sid) FROM Sailors S) + (SELECT COUNT (B.bid) FROM Boats B) < 100 )

CREATE TABLE Reserves
  ( name CHAR(10), 
  bid INTEGER, 
  day DATE, 
  PRIMARY KEY (bid,day), 
  CONSTRAINT minterlakesRes CHECK ( (SELECT B.name 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 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)

```sql
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.