SQL I
SQL – Introduction

- Standard DML/DDL for relational DB’s
  - DML = “Data Manipulation Language” (queries, updates)
  - DDL = “Data Definition Language” (create tables, indexes, …)

- Also includes:
  - view definition
  - security
  - integrity constraints
  - transactions

- History:
  - System R project at IBM: “SEQUEL”
  - later, becomes standard: Structured Query Language
Banking Example

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-other)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)
A Simple SELECT-FROM-WHERE Query

```
SELECT  bname
FROM     loan
WHERE   amt > 1000
```

<table>
<thead>
<tr>
<th>bname</th>
<th>lno</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>L-170</td>
<td>3000</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-230</td>
<td>4000</td>
</tr>
<tr>
<td>Perry</td>
<td>L-260</td>
<td>1700</td>
</tr>
<tr>
<td>Redwood</td>
<td>L-450</td>
<td>3000</td>
</tr>
</tbody>
</table>

Similar to:

\[ \pi_{\text{bname}} (\sigma_{\text{amt}>1000} (\text{loan})) \]

But not quite….

Why preserve duplicates?

- eliminating them is costly
- often, users don’t care
- can also write:

```
SELECT DISTINCT bname
FROM     loan
WHERE   amt > 1000
```
Another SFW query

```
SELECT cname, balance
FROM   depositor, account
WHERE  depositor.acct_no = account.acct_no

depositor (customer-name, account-number)
account (account-number, branch-name, balance)
```

Similar to:

```
π  cname, balance (depositor × account)
```

Note: you can also write

```
SELECT cname, balance
FROM   depositor AS d, account AS a
WHERE  d.acct_no = a.acct_no
```

<table>
<thead>
<tr>
<th>cname</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
<td>500</td>
</tr>
<tr>
<td>Smith</td>
<td>400</td>
</tr>
<tr>
<td>Turner</td>
<td>350</td>
</tr>
<tr>
<td>Smith</td>
<td>300</td>
</tr>
<tr>
<td>Jones</td>
<td>240</td>
</tr>
<tr>
<td>Smith</td>
<td>300</td>
</tr>
</tbody>
</table>
In general

SELECT A1, A2, …, An
FROM r1, r2, …, rm

[WHERE P]
  WHERE clause optional (missing WHERE clause means WHERE is true)

Conceptual Algorithm:

1. FROM clause: cartesian product ( X )
   • \( t1 \leftarrow r1 \times r2 \times … \times rm \)
2. WHERE clause: selection ( s )
   • \( t2 \leftarrow \sigma_p (t1) \)
3. SELECT clause: projection ( p )
   • result \( \leftarrow \pi_{A1, A2, …, An} (t2) \)

Note: will never be implemented with product \((X)\)!
The SELECT clause

equivalent to projection, despite name

can use “*” to get all attributes

  e.g.,  SELECT *
         FROM   loan

can write SELECT DISTINCT to eliminate duplicates

can write SELECT ALL to preserve duplicate (default)

can include arithmetic expressions

  e.g.,  SELECT bname, acct_no, balance*1.05
         FROM   account
The WHERE clause

- equivalent to selection, despite name…

- WHERE predicate can be:
  - Simple:
    - attribute relop attribute or constant
      (relop: <, >, =, ≤, ≥, ≠ )
  - Complex: using AND, OR, NOT, BETWEEN

- e.g.

```sql
SELECT lno
FROM loan
WHERE amt BETWEEN 9000 AND 10000
```

```sql
SELECT lno
FROM loan
WHERE amt >= 9000 AND amt <= 10000
```
RA can only express SELECT DISTINCT queries

To express SQL, must extend to a **bag algebra**, a bag (aka: multiset) like sets, but can have duplicates

e.g. \{ 4, 5, 4, 6\}

balances =

<table>
<thead>
<tr>
<th>cname</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
<td>500</td>
</tr>
<tr>
<td>Smith</td>
<td>400</td>
</tr>
<tr>
<td>Turner</td>
<td>350</td>
</tr>
<tr>
<td>Smith</td>
<td>300</td>
</tr>
<tr>
<td>Jones</td>
<td>240</td>
</tr>
<tr>
<td>Smith</td>
<td>300</td>
</tr>
</tbody>
</table>
The FROM clause

- Equivalent to cartesian product (X)
  (or ★ depending on the WHERE clause)

- binds tuples in relations to variable names
- e.g.: FROM borrower, loan
  - computes: borrower x loan
  - identifies borrower, loan columns (attrs) in the results

  e.g. allowing one to write:

  WHERE borrower.lno = loan.lno

FROM borrower b, loan l
WHERE b.lno = l.lno

• Simplifies the expression
• Needed for self-joins
Formal Semantics of SQL: RA*

1. $\sigma^*_P(r)$: preserves copies in $r$
   - $\sigma^*_\text{cname} = \text{"Smith" (balances)}$

   \begin{array}{|c|c|}
   \hline
   \text{cname} & \text{balance} \\
   \hline
   \text{Smith} & 400 \\
   \text{Smith} & 300 \\
   \text{Smith} & 300 \\
   \hline
   \end{array}

2. $\pi^*_A_1, A_2, \ldots, A_n(r)$: no duplicate elimination

   $\pi^*_\text{cname (balances)} =$

   \begin{array}{|c|c|}
   \hline
   \text{cname} & \text{balance} \\
   \hline
   \text{Johnson} & 500 \\
   \text{Smith} & 400 \\
   \text{Turner} & 350 \\
   \text{Smith} & 300 \\
   \text{Jones} & 240 \\
   \text{Smith} & 300 \\
   \hline
   \end{array}
Formal Semantics of SQL: RA*

3. $r \cup^* s$: additive union:

e.g. if $r = \begin{array}{c|c}
A & B \\
1 & a \\
1 & a \\
2 & b \\
\end{array}$, $s = \begin{array}{c|c}
A & B \\
2 & b \\
3 & a \\
1 & a \\
\end{array}$

then: $r \cup^* s =$

\begin{array}{c|c}
A & B \\
1 & a \\
1 & a \\
2 & b \\
2 & b \\
3 & a \\
1 & a \\
\end{array}$
Formal Semantics of SQL: RA*

4. \( r -* s : \text{bag difference} \)

e.g. \( r -* s = \)

\[
\begin{array}{cc}
A & B \\
1 & a \\
1 & a \\
2 & b \\
\end{array}
\]

\[
\begin{array}{cc}
A & B \\
2 & b \\
3 & a \\
1 & a \\
\end{array}
\]

\( s -* r = \)

\[
\begin{array}{cc}
A & B \\
3 & a \\
\end{array}
\]
5. $r \times^* s$: cartesian product with bags

E.g. if: $r = \begin{array}{cc}
A & B \\
1 & a \\
1 & a \\
1 & a \\
2 & b
\end{array}$, $s = \begin{array}{c}
C \\
\alpha \\
\gamma
\end{array}$

then: $r \times^* s = \begin{array}{ccc}
A & B & C \\
1 & a & \alpha \\
1 & a & \gamma \\
1 & a & \alpha \\
1 & a & \gamma \\
2 & b & \alpha \\
2 & b & \gamma
\end{array}$
Formal Semantics of SQL: RA*

Query
SELECT A1, A2, ..., An
FROM r1, r2, ..., rm
WHERE P

Semantics:
\[ \pi^*_{A1, A2, ..., An} (\sigma^*_P (r1 \times^* r2 \times^* ... \times^* rm)) \]

Query: SELECT DISTINCT A1, A2, ..., An
FROM r1, r2, ..., rm
WHERE P

Q: What is the only RA operator that need be changed above?

Ans: \( \pi^* \)
More SQL: Range variables

1. Using AS in **FROM** clause
   introduces tuple variables

   e.g.: 
   ```sql
   SELECT DISTINCT T.bname
   FROM branch AS T, branch AS S
   WHERE T.assets > S.assets
   ```

2. Using AS in **SELECT** clause
   renames columns in result (ρ)

   e.g.: 
   ```sql
   SELECT bname, acct_no, balance*1.05 AS newbal
   FROM account
   ```

<table>
<thead>
<tr>
<th>bname</th>
<th>acct_no</th>
<th>newbal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-170</td>
<td>450</td>
</tr>
<tr>
<td>Redwood</td>
<td>A-230</td>
<td>400</td>
</tr>
</tbody>
</table>
More SQL: Intro

Give a name to a query result (←)

E.g.

```
SELECT DISTINCT bname
INTO BranchNames
FROM branch
```

intuitively: BranchNames ← SELECT DISTINCT bname
FROM branch
More SQL - String Operations

- SQL includes a string-matching operator
- percent (%). The % character matches any substring.
- underscore (_). The _ character matches any character.
- E.g. Find the names of all customers whose street includes the substring “Main”.

```sql
SELECT customer-name
FROM customer
WHERE cstreet LIKE ‘%Main%’
```

match the name “Main%”:
use:    `like ‘Main\%’`    `escape ‘\’`

SQL supports a variety of string operations such as
  • concatenation (using “||”)
  • converting from upper to lower case (and vice versa)
  • finding string length, extracting substrings, etc.
More SQL: Set/Bag operations

Set operations:
    UNION, INTERSECT, EXCEPT (MINUS)

Bag operations:
    UNION ALL, INTERSECT ALL, EXCEPT ALL

Duplicate counting:
    Given m copies of α in r and n copies of α in s

Q: How many copies of α in ....

1. r UNION ALL s
   1. Ans: m + n

2. r INTERSECT ALL s
   2. Ans: min (m,n)

3. r EXCEPT ALL s
   3. Ans: max(0, m-n)
More SQL: Set/Bag operations

Example Queries:

(SELECT c
name FROM depositor)

? = UNION
• returns names of customers with saving accts, loans, or both

(SELECT c
name FROM borrower)

? = INTERSECT
• returns names of customers with saving accts AND loans

? = EXCEPT
• returns names of customers with saving accts but NOT loans
Example: List in alphabetical order, the names of all customers with loans at Kenmore branch:

```
SELECT DISTINCT cname
FROM borrower b, loan l
WHERE b.lno = l.lno AND bname = 'Kenmore'
ORDER BY cname
```

Result: 

<table>
<thead>
<tr>
<th>Cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
</tr>
<tr>
<td>Byers</td>
</tr>
<tr>
<td>Smith</td>
</tr>
</tbody>
</table>

Can also write:

```
ORDER BY cname DESC or
ORDER BY cname ASC (default)
```

Like: SELECT DISTINCT, very expensive...
Aggregate Operators

- **Aggregate Operators:**
  - AVG (col): average of values in col
  - MIN (col): minimum value in col
  - MAX (col): maximum value in col
  - SUM (col): sum of values in col
  - COUNT (col): number of values in col

Examples:
1. Find the average acct balance @ Allston:
   ```sql
   SELECT AVG (bal)
   FROM account
   WHERE bname = 'Allston'
   ```
2. Find the number of tuples in customer:
   ```sql
   SELECT COUNT(*)
   FROM customer
   ```
3. Find the number of depositors
   ```sql
   SELECT COUNT( DISTINCT cname)
   FROM depositor
   ```

COUNT, SUM, AVG have a DISTINCT version.
Aggregates and Group By

- Usually, aggregates used with “Group By”
- E.g.

```sql
SELECT    bname, COUNT (DISTINCT cname)
FROM       depositor d, account  a
WHERE      d.acct_no = a.acct_no
GROUP BY   bname
```

Result: 

<table>
<thead>
<tr>
<th>bname</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>2</td>
</tr>
<tr>
<td>Mianus</td>
<td>5</td>
</tr>
<tr>
<td>Perry</td>
<td>1</td>
</tr>
<tr>
<td>Brighton</td>
<td>5</td>
</tr>
<tr>
<td>Kenmore</td>
<td>7</td>
</tr>
</tbody>
</table>

*depositor (customer-name, account-number)*

*account (account-number, branch-name, balance)*
Aggregates and Group By

Intuition behind “Group By”

```
SELECT    bname, COUNT (DISTINCT cname)
FROM       depositor d, account  a
WHERE      d.acct_no = a.acct_no
GROUP BY   bname
```

Step 1: “Group “ results of join

<table>
<thead>
<tr>
<th>bname</th>
<th>a.acct_no</th>
<th>balance</th>
<th>cname</th>
<th>d.acct_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
<td>Johnson</td>
<td>A-101</td>
</tr>
<tr>
<td>Mianus</td>
<td>A-215</td>
<td>700</td>
<td>Smith</td>
<td>A-215</td>
</tr>
<tr>
<td>Perry</td>
<td>A-102</td>
<td>400</td>
<td>Hayes</td>
<td>A-102</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-202</td>
<td>900</td>
<td>Johnson</td>
<td>A-202</td>
</tr>
<tr>
<td>Brighton</td>
<td>A-217</td>
<td>800</td>
<td>Jones</td>
<td>A-217</td>
</tr>
<tr>
<td>Kenmore</td>
<td>A-305</td>
<td>700</td>
<td>Smith</td>
<td>A-305</td>
</tr>
<tr>
<td>Kenmore</td>
<td>A-232</td>
<td>600</td>
<td>Lindsay</td>
<td>A-232</td>
</tr>
</tbody>
</table>

Step 2: aggregate on groups and project on result

<table>
<thead>
<tr>
<th>lname</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>1</td>
</tr>
<tr>
<td>Mianus</td>
<td>1</td>
</tr>
<tr>
<td>Perry</td>
<td>1</td>
</tr>
<tr>
<td>Brighton</td>
<td>2</td>
</tr>
<tr>
<td>Kenmore</td>
<td>2</td>
</tr>
</tbody>
</table>
Another example:

```
branch(bname, bcity, assets)
```

```
SELECT bname, SUM(assets) as total
FROM branch
GROUP BY bcity
```

Result ??

<table>
<thead>
<tr>
<th>bname</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redwood</td>
<td>2.1M</td>
</tr>
<tr>
<td>Pownal</td>
<td>0.3M</td>
</tr>
<tr>
<td>N. Town</td>
<td>3.7M</td>
</tr>
</tbody>
</table>

Above query is NOT allowed

Non-aggregated values in SELECT clause (e.g., bname) must also appear in GROUP BY clause

```
SELECT A1, A2, ..., Ak, Agg1(), ...., Aggi()
FROM ..........
WHERE ..........
GROUP BY A1, A2, ..., Ak, Ak+1, ..., An
```
HAVING

WHERE :: FROM    as HAVING :: GROUP BY

➢ HAVING P: selects rows from result of GROUP BY
➢ Optional (missing HAVING means TRUE)

Example: Find names of branches and the average account balance for those branches having average account balance > $1200

```sql
SELECT  bname, AVG(balance) AS avg
FROM     account
GROUP BY bname
HAVING  avg > 1200
```

Result same as:

```sql
SELECT  bname, AVG(balance) AS avg
INTO        Temp
FROM     account
GROUP BY bname

SELECT  *
FROM      Temp
WHERE     avg > 1200
```
Find sailors who’ve reserved at least one boat

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

- 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?
Find sid’s of sailors who’ve reserved a red or a green boat

- **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).

```sql
SELECT R.sid
  FROM Boats B, Reserves R
 WHERE R.bid=B.bid AND
   (B.color='red' OR B.color='green')
```

Vs.

```sql
SELECT  R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
UNION
SELECT R.sid
  FROM Boats B, Reserves R
 WHERE R.bid=B.bid AND B.color='green'
```
Find sid’s of sailors who’ve reserved a red but did not reserve a green boat

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
    AND R.bid=B.bid
    AND B.color='red'
EXCEPT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid
    AND R.bid=B.bid
    AND B.color='green'
```
**NULLs**

The “dirty little secret” of SQL (major headache for query optimization) can be a value for any attribute

e.g.:

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horse</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horse</td>
<td>.4M</td>
</tr>
<tr>
<td>Kenmore</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

What does this mean?

- We don’t know Kenmore’s assets?
- Kenmore has no assets?
- ....................

**Effect on Queries:**

```sql
SELECT * FROM branch2
WHERE assets = NULL
```

```sql
<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kenmore</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

```sql
SELECT * FROM branch2
WHERE assets  IS NULL
```

```sql
<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kenmore</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```
NULLs

- Arithmetic with nulls:
  - \( n \text{ op null} = \text{null} \)
    - \( \text{op: +, -, *, /, mod, ...} \)

- Booleans with nulls: One can write:
  3-valued logic (true, false, unknown)

SELECT .......... FROM .......... WHERE boolexpr IS UNKNOWN

What expressions evaluate to UNKNOWN?
1. Comparisons with NULL (e.g. assets = NULL)
2. FALSE OR UNKNOWN (but: TRUE OR UNKNOWN = TRUE)
3. TRUE AND UNKNOWN
4. UNKNOWN AND/OR UNKNOWN
NULLs

Given:

<table>
<thead>
<tr>
<th>bname</th>
<th>bcity</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Boston</td>
<td>9M</td>
</tr>
<tr>
<td>Perry</td>
<td>Horse</td>
<td>1.7M</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horse</td>
<td>.4M</td>
</tr>
<tr>
<td>Kenmore</td>
<td>Boston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Aggregate operations:

```
SELECT SUM(assets)
FROM branch2
```
returns SUM

\[ 11.1M \]

NULL is ignored
Same for AVG, MIN, MAX

But.... COUNT(assets) returns 4!

Let branch3 an empty relation
Then: SELECT SUM(assets)

FROM branch3
returns NULL
but COUNT(<empty rel>) = 0
Review - Summary Thus Far

Kitchen sink query:

```sql
SELECT bcity, sum(balance) AS totalbalance
INTO BranchAcctSummary
FROM branch b, account a
WHERE b.bname=a.bname AND assets >= 1M
GROUP BY bcity
HAVING totalbalance > 700
ORDER BY bcity DESC
```

Steps 1,2: FROM, WHERE

<table>
<thead>
<tr>
<th>b.bname</th>
<th>bcity</th>
<th>assets</th>
<th>a.bname</th>
<th>acct_no</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downtown</td>
<td>Bkln</td>
<td>9M</td>
<td>Downtown</td>
<td>A-101</td>
<td>500</td>
</tr>
<tr>
<td>Redwood</td>
<td>Palo Alto</td>
<td>2.1M</td>
<td>Redwood</td>
<td>A-215</td>
<td>700</td>
</tr>
<tr>
<td>Perry</td>
<td>Horse</td>
<td>1.7M</td>
<td>Perry</td>
<td>A-102</td>
<td>400</td>
</tr>
<tr>
<td>RH</td>
<td>Horse</td>
<td>8M</td>
<td>RH</td>
<td>A-202</td>
<td>350</td>
</tr>
<tr>
<td>Brighton</td>
<td>Bkln</td>
<td>7.1M</td>
<td>Brighton</td>
<td>A-305</td>
<td>900</td>
</tr>
<tr>
<td>Brighton</td>
<td>Bkln</td>
<td>7.1M</td>
<td>Brighton</td>
<td>A-217</td>
<td>750</td>
</tr>
</tbody>
</table>
Steps 3, 4: GROUP BY, SELECT

<table>
<thead>
<tr>
<th>bcity</th>
<th>totalbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bkln</td>
<td>2150</td>
</tr>
<tr>
<td>Palo Alto</td>
<td>700</td>
</tr>
<tr>
<td>Horse</td>
<td>750</td>
</tr>
</tbody>
</table>

Steps 5: HAVING

<table>
<thead>
<tr>
<th>bcity</th>
<th>totalbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bkln</td>
<td>2150</td>
</tr>
<tr>
<td>Horse</td>
<td>750</td>
</tr>
</tbody>
</table>

Steps 6: ORDER BY

<table>
<thead>
<tr>
<th>bcity</th>
<th>totalbalance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horse</td>
<td>750</td>
</tr>
<tr>
<td>Bkln</td>
<td>2150</td>
</tr>
</tbody>
</table>

Steps 7: INTO
## Summary thus far

<table>
<thead>
<tr>
<th>Clause</th>
<th>Evaluation Order</th>
<th>Semantics (RA/RA*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT[DISTINCT]</td>
<td>4</td>
<td>$\pi^*$ (or $\pi$)</td>
</tr>
<tr>
<td>FROM</td>
<td>1</td>
<td>$X^*$</td>
</tr>
<tr>
<td>WHERE</td>
<td>2</td>
<td>$\sigma^*$</td>
</tr>
<tr>
<td>INTO</td>
<td>7</td>
<td>$\leftarrow$</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>3</td>
<td>Can’t express</td>
</tr>
<tr>
<td>HAVING</td>
<td>5</td>
<td>$\sigma^*$</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>6</td>
<td>Can’t express</td>
</tr>
</tbody>
</table>