Background (1 / 2)

- IBM’s System R was released in 1978
  - Its query language name: SEQUEL
    (Structured English QUEry Language)
  - But trademarked by a British airplane company!
    (1982, Hawber Siddeley Dynamics Engineering Ltd.)
  - After dropping the vowels: SQL
- IBM’s current DB/2 was released in 1982; also used SQL
- SQL:
  - A marriage of TRC to RA
  - SQL = DML + DDL + DCL + QL + . . .
Background (2 / 2)

- SQL is no longer a proprietary language:
  - SQL is now an ANSI/ISO standard (ISO/IEC 9075)
- But no DBMS strictly follows any of them!
  - There is a basic subset you can count on
  - Example: Tuple IDs are non-standard

Relational Operators (1 / 5)

But first: SQL’s SELECT statement

- NOT identical to the select operator of Rel. Alg.!
- Most basic Form:

  ```sql
  SELECT <attribute list>
  FROM <relation list>;
  ```

Example(s):
Relational Operators (2 / 5)

Now that we can perform $\pi$, we can answer our first standard query:

“What is the content of the Employee relation?”

Relational Operators (3 / 5)

Performing $\sigma$ requires a new clause:

```
SELECT <attribute list>
FROM   <relation list>
```

Example(s):

What are the names and salaries of employees in department 5?
These are also all of the clauses that we need for $\bowtie$:

**Example(s):**

What are the names of the parts that can be supplied by individual suppliers in quantity $> 200$?

For completeness, our fourth standard query:

**Example(s):**

What are the names of the active suppliers of nuts?

```sql
SELECT sname
FROM s, spj, p
WHERE s.sno = spj.sno
  AND spj.pno = p.pno
  AND status > 0
  AND pname = 'nut';
```
Renaming Attributes

You may give your result relations new attribute names:

Example(s):

```
select givenname as "First Name",
surname as "Last Name",
salary
from employee
where deptid = 5;
```
Ordering Result Tuples

We can sort tuples, too, with the ORDER BY clause.

Example(s):

Computed Columns

We can perform basic arithmetic with field values:

Example(s): Convert part weight from pounds to grams:
Tuple Variables (a.k.a. Aliases)

We can assign relations temporary, alternate names.

Example(s):

Create all pairs of supplier names located within the same city:

Pattern Matching (1 / 2)

SQL allows us to search for values that match a particular pattern.

Form:

\[
\ldots \text{WHERE } \text{attribute [not] LIKE} \, \text{'pattern'}
\]

[ESCAPE \text{escape character}]

Available wildcards:

\%

matches 0 or more characters

_ (underscore) matches any single character

Important: LIKE does not support regular expressions.
Find the part names that have an ‘o’ as the second letter:

```sql
select pname
from p
where pname like '__o%';
```

To use wildcards as regular characters, ESCAPE them:

```sql
... where field like '%@%' escape '@';
```

Here, we match any string ending in a percent sign.

Oracle offers REGEXP LIKE for regular expressions.

Form (note that `<pattern>` and `<match>` are single–quoted):

```sql
... WHERE REGEXP LIKE ( <source>, '<pattern>', '<match>' );
```

where:

- `<source>` is an attribute name
- `<pattern>` is a regular expression (see next slide)
- `<match>` is a search modifier; e.g.:
  - `c` — case sensitive
  - `i` — case insensitive
  - `x` — ignore whitespace
  - `:`
Regular Expressions (2 / 2)

REGEXP LIKE options for <pattern> include:

- . — (a period) match a single character
- x* — match x 0 or more times
- x+ — match x 1 or more times
- x? — match x 0 or 1 times
- x|y — match x once or match y once
- x{n,m} — match x at least n times, at most m times

Example(s):

Find the part names that have an ‘o’ as the second letter:

```sql
select pname
from p
where regexp_like( pname , '\.o()*' , 'i' );
```

Set Operators (1 / 5)

Cartesian Product (×):

- Cartesian Product produces all pairs of tuples.
- Join produces all pairs of tuples that meet a condition.
- So ... if we Join when the condition is always true ...

Example(s): To form the Cartesian Product of S and P:
Set Operators (2 / 5)

Union (∪):

- Form: select . . .
  union [all] (all ≡ keep duplicates)
  select . . .
- Union compatibility still applies!

Example(s):

Set Operators (3 / 5)

Intersection (∩) and Difference (−):

- The SQL keyword for set intersection is INTERSECT
- The SQL keyword for set difference is EXCEPT . . . except, Oracle uses MINUS
- Form: select . . .
  intersect/except
  select . . .

Example(s):

```sql
select city from s
EXCEPT
<- -- MINUS in Oracle
select city from p;
```
Set Operators (4 / 5)

The Return of . . . Division!

Version 1: Relational Algebra expression

Recall: \[ \alpha \div \beta = \pi_{A-B}(\alpha) - \pi_{A-B}((\pi_{A-B}(\alpha) \times \beta) - \alpha) \]

And our sample division query:

“Find the S#s of the suppliers who supply all parts of weight equal to 17.”

Set Operators (5 / 5)

And so, \[ \alpha \div \beta = \pi_{A-B}(\alpha) - \pi_{A-B}((\pi_{A-B}(\alpha) \times \beta) - \alpha) \]

becomes in SQL:

```sql
select distinct sno from spj
except
select sno from
  ( select sno, pno
    from (select sno from spj) as t1,
         (select pno from p where weight=17) as t2
    except
      select sno, pno from spj
  ) as t3;
```
Aggregate Functions (1 / 3): Background

Idea: Let SQL compute basic statistical results for us

SQL provides aggregate functions for this purpose:

- `count([distinct] attr)` — counting entries in a relation
- `sum([distinct] attr)` — totaling values of `attr` in a relation
- `avg([distinct] attr)` — averaging values of `attr` in a relation
- `min(attr)` — smallest value of `attr` in a relation
- `max(attr)` — largest value of `attr` in a relation

Aggregate Functions (2 / 3)

Example(s): Variations on counting:

- `select count(city) from p;`
- `select count(distinct city) from p;`
- `select count(*) from p;`
Example(s):

If we have one of each part in a box, how much does the content weigh?

*Which query will give the correct answer?*

(a) `select sum(weight) from p;`
(b) `select sum(distinct weight) from p;`

---

Group By

**Purpose:** Apply aggregates to sub–groups of tuples

**Example(s):**

What are the average quantities in which suppliers are supplying parts?
Having

- Used in conjunction with ‘group by’
- Purpose: Controls which group’s aggregations are produced

Example(s):

Which suppliers are supplying parts in average quantity under 400, and what are those averages?

More on Nested Queries (1 / 4)

We’ve seen this idea before (e.g., the division query).

Another way to do nested queries is with the IN operator:

- IN tests set membership (form: tuple IN relation)
- We can negate the test (tuple NOT IN relation)
- Used in conjunction with a sub-query in a WHERE clause

Example(s):

Remember this query?
More on Nested Queries (2 / 4)

Example(s):

Idea: Create a set of parts available in quantity > 200, and test each part from the DB against that set.

To create the P#s of the ‘quantity > 200’ parts:

```sql
select pno
from spj
where qty > 200;
```

And to produce the names of the parts in that set:

More on Nested Queries (3 / 4)

Notes:

- IN and NOT IN are only suitable for equality comparisons
- Other options include:
  -
  -
  -
  -
  -
More on Nested Queries (4 / 4)

One more nested–query operator: EXISTS

Its purpose: Test if a relation holds at least one tuple

Example(s):

Another (awkward!) version of the qty > 200 query:

Division, Revisited (1 / 7)

Version 2: “Double ¬ ∃”

Recall:

Find the S#s of the suppliers who supply all parts of weight 17.

Restated in logical English:

Find S#s such that ∀ parts of weight 17, ∃ suppliers that supply them all

Apply Double Negation and Generalized De Morgan’s Laws:

∀a ∃b f(a, b) ≡ ¬ ∃a ¬ ∃b f(a, b)

Returning to logical English:

Find S#s such that ¬ ∃ parts of weight 17 for which ¬ ∃ suppliers that supply them all
Find S#s such that ¬ ∃ parts of weight 17 for which ¬ ∃ suppliers that supply them all expressed in SQL:

```sql
select distinct sno
from spj as global
where not exists
  ( select pno
      from p
      where weight = 17 and not exists
        ( select *
          from spj as local
          where local.pno = p.pno
            and local.sno = global.sno
        )
  )
```

Aside: This query form is useful beyond division.

Example(s): Which drinkers like a unique set of beers?

Observation:

If $B \subseteq A$, then $B - A$ will be empty (or, $\neg \exists (B - A)$)

Relevance:

If a supplier supplies a superset of the parts of weight 17, the supplier clearly supplies them all

$A = \text{The parts a supplier supplies}$

$B = \text{The parts of weight 17}$

```sql
select distinct sno
from spj as global
where not exists ( -- not bkwd-E
    ( select pno
        from p -- B
        where weight = 17
    ) except ( -- minus
        select p.pno
        from p, spj -- A
        where p.pno = spj.pno
            and spj.sno = global.sno
    )
)
```
Version 4: Set Cardinality

Idea:

- For each supplier that supplies parts of weight 17, count those parts.
- If the total matches the number of weight 17 parts, that supplier supplies them all.

```sql
select distinct sno
from spj, p
where spj.pno = p.pno and weight = 17
group by sno
having count(distinct p.pno) =
    ( select count (distinct pno)
        from p
        where weight = 17
    )
```
Outer Joins (1 / 5)

Regular (“inner”) joins discard non-matching tuples.

**Example(s):** Name the employees who are supervising buildings.

<table>
<thead>
<tr>
<th></th>
<th>Id</th>
<th>Name</th>
<th>Building</th>
<th>Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Roy</td>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Amy</td>
<td></td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Joy</td>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Now consider this slightly different query.

**Example(s):** Name all employees and the buildings they supervise.

<table>
<thead>
<tr>
<th></th>
<th>Id</th>
<th>Name</th>
<th>Building</th>
<th>Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Roy</td>
<td></td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Amy</td>
<td></td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Amy</td>
<td></td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Joy</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

But … how do we get this result from a join?
Outer Joins (3 / 5)

There are three varieties of outer join:

- Left Outer Join (\(\sqsupset\leftarrow\rightarrow\)): Retains unmatched tuples from left relation
- Right Outer Join (\(\leftarrow\rightarrow\sqsubset\)): Retains unmatched tuples from right relation
- Full Outer Join (\(\sqsupset\leftarrow\rightarrow\sqsubset\)): Retains all unmatched tuples

Outer Joins (4 / 5)

The SQL outer join syntax:

```sql
select <attribute list>
from ( <relation> [left/right/full] outer join <relation> on <join condition> )
where <condition> ;
```

**Example(s):** Name all employees and the buildings they supervise.
Outer Joins (5 / 5)

Outer join is not an fundamental operator.

We can fabricate outer join with UNION ALL.

Example(s): Name all employees and the buildings they supervise.

---

SQL as DDL

First order of business: Creating a database!

The exact mechanism depends on the DBMS.

1. Postgres: $ createdb <name>
2. Oracle: CREATE DATABASE <name>;
Creating Relations (1 / 3)

Some sample attribute types:

- **Integers**: integer, number(p)
- **Floats**: float, real, number(p, s)
  - p is precision (total # digits), s is scale (# digits after decimal)
- **Strings**: char(n), varchar(n), varchar2(n)
- **Others**: timestamp, blob, bfile, ...

Creating Relations (2 / 3)

To create a relation:

```
CREATE TABLE <table name> (  
  <attribute name> <data type> [ NOT NULL ],
  . . .
  [ PRIMARY KEY ( <attribute> ) ]
);
```
Creating Relations (3 / 3)

Example(s):

Creating the supplier (S) relation:

```sql
create table s (
    sno    varchar2(5), -- the supplier ID number
    sname  varchar2(20), -- the supplier’s name
    status integer,     -- supplier status
    city   varchar2(15), -- location of supplier
    primary key (sno)
);
```

Creating Indices (1 / 3)

Form:

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table name>
[ USING <access method> ]
( <attribute name> [, <attribute name> ... ]);
```
Example(s):

Create an index on `jno` in SPJ:

```sql
create index spj_j_index on spj (jno);
```
Creating Views (1 / 2)

Remember the ANSI/SPARC External Layer?

Form:

```
CREATE VIEW <view name> [ ( <attribute list> ) ]
AS <select statement>;
```

Creating Views (2 / 2)

Example(s):

Create a view of supplier names and the IDs of the parts that they supply.

```
create view supplierpart ("Supplier Name", "Part ")
as select distinct sname, pno
    from  s, spj
    where s.sno = spj.sno;
```

Then, it is available for use:

```
select * from supplierpart;
```
Can users update the content of views? That is, can we convert a view update into updates of the view's base relations?

**Example(s):**

Consider a view that is a join of A and B:

<table>
<thead>
<tr>
<th>A</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
<td>b</td>
<td>6</td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>b</td>
<td>y</td>
</tr>
</tbody>
</table>

A ⊲ ⊳ B

<table>
<thead>
<tr>
<th>A ⊲ ⊳ B</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>c</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

⇒

<table>
<thead>
<tr>
<th>A ⊲ ⊳ B</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>c</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>y</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>x</td>
<td>2</td>
<td>c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>y</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
</tr>
<tr>
<td>4</td>
<td>y</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
</tr>
</tbody>
</table>

**Example(s):** (continued!) Our desired result:
The view update example raises a pertinent question:

**How do we insert data into a relation?**

With a DML operation, of course!

---

**Inserting Tuples into a Relation**

To insert a tuple into a relation:

```
INSERT INTO <relation name> [ ( <column list> ) ]
VALUES ( <expression list> );
```

**Example(s):**
Bulk Loading a Database

Using INSERT INTO to populate tables is:

- **Highly portable!** (just create a script file), but
- **Slow** (especially you don’t disable transations)

An alternative is a bulk–loading utility.

**Example(s):**

---

Updating Content of Tuples

To modify data in existing tuples:

```
UPDATE <relation name>

SET <attribute name> = <expression> [, . . . ]

[ FROM <relation list> ]

[ WHERE <condition> ];
```

**Example(s):**

---
Deleting Tuples

Like updating, a condition is used to ID tuples for removal:

DELETE FROM <relation name>
WHERE <condition>;

Deleting Relations

To remove tables, indices, views, . . .

DROP { TABLE | INDEX | VIEW | DATABASE } <name>;
Storing Query Results

Can we add query results (which are relations) to the DB?

Yes! Two options:

1. (Pretty universal) If you have an existing table:

   \[
   \text{INSERT INTO} \quad <\text{relation name}> \\
   \quad <\text{SELECT statement}>; \\
   \]

2. (Oracle) If you need to create the table, too:

   \[
   \text{CREATE GLOBAL TEMPORARY TABLE} \quad <\text{relation name}> \\
   \quad \text{AS} \quad <\text{SELECT statement}>; \\
   \]

   (Table disappears at end of session.)

Wait! What About “SQL as DCL?”

We’ll cover that in Topic 14: Security.