Why Formal languages?
Example: Optimization Is Based on Algebraic Equivalences

- Relational algebra has laws of commutativity, associativity, etc. that imply certain expressions are equivalent in semantics
- They may be different in cost of evaluation!
  - $\sigma_{p1 \land p2}(R) = \sigma_{p1}(\sigma_{p2}(R))$
  - $(R1 \bowtie R2) = (R2 \bowtie R1)$
  - $(R1 \bowtie R2) \bowtie R3 = R1 \bowtie (R2 \bowtie R3)$

△ Query optimization finds the most efficient representation to evaluate (or one that’s not bad)

The Big Picture: SQL to Algebra to Query Plan to Web Page

SQL
- Based primarily on Relational Algebra with some features from Relational Calculus
- Components: Data definition language, Manipulation language
- Other SQL features
  - Transaction definition: end of query by default
  - Security, Views, Index
- Embedded SQL – embed SQL commands in a general purpose language
- Database connectivity packages allow queries to be passed to DB from applications – JDBC
Basic SQL Query

SELECT [DISTINCT] attribute-list
    FROM relation-list
    WHERE qualification/predicate

- relation-list: A list of relation names (possibly with a range-variable, i.e., tuple variable, after each name).
- attribute-list: A list of attributes of relations in relation-list
- Qualification/predicate: Comparisons (Attr op const or Attr1 op Attr2, where op is one of \(<\), \(\geq\), \(\leq\) ) 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!

SQL Queries

- Querying with SQL is performed using a SELECT statement.
- The general form of the statement is:
  
  SELECT A_1, A_2, ..., A_n attributes in result
  FROM R_1, R_2, ..., R_m tables in query
  WHERE (condition)

- Notes:
  - 1) The "*" is used to select all attributes.
  - 2) Combines the relational algebra operators of selection, projection, and join into a single statement.
  - 3) Comparison operators: =, !=, >, <, >=, <=.

SQL and Relational Algebra

- The SELECT statement can be mapped directly to relational algebra.

  SELECT A_1, A_2, ..., A_n /* this is projection
  FROM R_1, R_2, ..., R_m /* this is the selection op
  WHERE P /* this is cartesian product

  is equivalent to:

  \[ \Pi_{A_1, A_2, ..., A_n}(\sigma_P(R_1 \times R_2 \times ... \times R_m)) \]

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
- Compute the cross-product of relation-list.
- Discard resulting tuples if they fail predicate qualifications.
- Delete attributes that are not in target attribute-list.
  - If DISTINCT is specified, eliminate duplicate rows.
- SQL allows duplicates in relations (unlike Rel. Algebra)
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute the same answers.
One Relation Query Example

- Return the employee name and salary of all employees whose title is 'EE':

\[
\text{SELECT ename, salary FROM emp WHERE title = 'EE'}
\]

<table>
<thead>
<tr>
<th>Emp Relation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>eno</td>
<td>ename</td>
</tr>
<tr>
<td>E1</td>
<td>J. Doe</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
</tr>
<tr>
<td>E6</td>
<td>L. Chu</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
</tr>
</tbody>
</table>

Algorithm: Scan each tuple in table and check if matches condition in WHERE clause.

Join Query Example

- Multiple tables can be queried in a single SQL statement by listing them in the FROM clause.
  - Note that if you do not specify any join condition to relate them in the WHERE clause, you get a cross product of the tables.

- Example: Return the employees who are assigned to the 'Management' department.

\[
\text{SELECT ename FROM emp, dept WHERE dname = 'Management' and emp.dno = dept.dno}
\]

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ename</td>
</tr>
<tr>
<td>R. Davis</td>
</tr>
<tr>
<td>J. Jones</td>
</tr>
</tbody>
</table>
Join Query Examples

Return the employees and the names of their department:

```
SELECT ename, dname
FROM emp, dept
WHERE emp.dno=dept.dno
```

Return all projects who have an employee working on them whose title is 'EE':

```
SELECT pname
FROM emp, proj, workson
WHERE emp.title = 'EE' and workson.eno=emp.eno
and workson.pno = proj.pno
```

SQL….review basic operations

- Select clause
- Need to specify Join condition
- Concept of 'aliasing' to rename relation using AS keyword
  - Rename an attribute….in Select clause
  - Rename a relation
- Important: can leave out the AS keyword!

- INTO clause
  - Variation on aliasing
  - Pipe output of SELECT into another table

Aliasing: INTO clause

- Often it is useful to ‘route’ output of Select into another (temp) table…accomplished using the keyword INTO:

```
SELECT ename, pname, salary AS pay
INTO NewTable
FROM emp, workson, proj
WHERE emp.eno = workson.eno and ename='A. Lee'
and proj.pno = workson.pno
```

```
<table>
<thead>
<tr>
<th>ename</th>
<th>pname</th>
<th>pay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Lee</td>
<td>Budget</td>
<td>192.31</td>
</tr>
<tr>
<td>A. Lee</td>
<td>Maintenance</td>
<td>923.08</td>
</tr>
</tbody>
</table>
```

- Can then write another query:

```
SELECT *
FROM NewTable;
```

Aliasing: rename an attribute

- Often it is useful to be able to rename an attribute in the final result (especially when using calculated fields). Renaming is accomplished using the keyword AS:

```
SELECT ename, pname, salary AS pay
FROM emp, workson, proj
WHERE emp.eno = workson.eno and ename='A. Lee'
and proj.pno = workson.pno
```

```
<table>
<thead>
<tr>
<th>ename</th>
<th>pname</th>
<th>pay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Lee</td>
<td>Budget</td>
<td>192.31</td>
</tr>
<tr>
<td>A. Lee</td>
<td>Maintenance</td>
<td>923.08</td>
</tr>
</tbody>
</table>
```

Result

```
<table>
<thead>
<tr>
<th>ename</th>
<th>pname</th>
<th>pay</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Lee</td>
<td>Budget</td>
<td>192.31</td>
</tr>
<tr>
<td>A. Lee</td>
<td>Maintenance</td>
<td>923.08</td>
</tr>
</tbody>
</table>
```

Note: AS keyword is optional.
Renaming and Aliasing (2)

- Renaming is also used when two or more copies of the same table are in a query. Using aliases allows you to uniquely identify what table you are talking about.
  - Similar to relational algebra operator
Example: Return the employees and their managers where the managers make less than the employee.

```sql
SELECT E.ename, M.ename
FROM emp as E, emp as M
WHERE E.supereno = M.eno and E.salary > M.salary
```

- Important: There is another (more elegant) interpretation that accomplishes the same using concept of "tuples" variables
  - Taken from relational calculus

Aliasing/Renaming and Tuple Variables

- Aliasing is nothing but use of tuple variables borrowed from relational calculus
- Concept of tuple/range variables in rel. calculus
  - Tuple t of type R: t ∈ R
  - What about x ∈ R, y ∈ R
  - One variable with name x and one with name y
  - Both have the same 'type'

- Need to worry about scope of tuple variables when we have nested queries

Meaning (Semantics) of SQL Queries with tuples

```sql
SELECT a_1, a_2, ..., a_k
FROM R_1 x_1, R_2 x_2, ..., R_n x_n
WHERE Conditions
```

Answer = {} for x_1 in R_1 do
  for x_2 in R_2 do
    ..... for x_k in R_k do
      if Conditions
        then Answer = Answer ∪ {(a_1,....,a_k)}
  return Answer

Tuple variables

- Find customers (ID) who have an account at a branch where CustID 6666 has an account.
- Need to access Deposit table twice
  - Once to extract branch X of CustID 6666
  - Second time to find accounts at these branches X
  - Define two "variables" A,B of 'type' Deposit
  - B is variable that corresponds to CustID 6666 and its branch-name field is equal to "X"
  - A is a variable whose branch-name is equal to "X"
Tuple variables
- Find customers (ID) who have an account at a branch where CustID 6666 has an account.

```
SELECT A.CustID
FROM Deposit A, Deposit B
WHERE B.CustID=6666
    AND A.branchname=B.branchname
```
- B is a variable that refers to Customer 6666
- Variable A can take on any values that satisfy the query
- The type of A is schema of Deposit

Advanced Conditions - BETWEEN
- Sometimes the condition in the WHERE clause will request tuples where one attribute value must be in a range of values.
- Example: Return the employees who make at least $20,000 and less than or equal to $45,000.

```
SELECT ename
FROM emp
WHERE salary >= 20000 and salary <= 45000
```
- We can use the keyword BETWEEN instead:

```
SELECT ename
FROM emp
WHERE salary BETWEEN 20000 and 45000
```

Advanced Conditions - LIKE
- For string valued attributes, the LIKE operator is used to search for partial matches.
  - Partial string matches specified using either "%" that replaces an arbitrary number of characters or underscore "_" that replaces a single character.
- Example: Return all employee names that start with 'A'.

```
SELECT ename
FROM emp
WHERE ename LIKE 'A%'
```
- Example: Return all employee names who have a first name that starts with 'J' and whose last name is 3 characters long.

```
SELECT ename
FROM emp
WHERE ename LIKE 'J. _ _'
```

Performance Concerns of LIKE
- **Warning**: Do not use the LIKE operator if you do not have to.
- It is often an inefficient operation as the DBMS may not be able to optimize lookup using LIKE as it can for equal (=) comparisons. The result is the DBMS often has to examine ALL TUPLES in the relation.
- In almost all cases, adding indexes will **not** increase the performance of LIKE queries because the indexes cannot be used.
  - Most indexes are implemented using B-trees that allow for fast equality searching and efficient range searches.
**Advanced Conditions - IN**

- To specify that an attribute value should be in a given set of values, the `IN` keyword is used.
  - Example: Return all employees who are in any one of the departments {'D1', 'D2', 'D3'}.
    ```sql
    SELECT ename
    FROM emp
    WHERE dno IN ('D1','D2','D3')
    ```
  - Note that this is equivalent to using OR:
    ```sql
    SELECT ename
    FROM emp
    WHERE dno = 'D1' OR dno = 'D2' OR dno = 'D3'
    ```
  - However, more practical uses of `IN` and `NOT IN` when we study nested subqueries.

**Advanced Conditions - NULL**

- Remember NULL is used to indicate that a given attribute does not have a value. To determine if an attribute is NULL, we use the clause `IS NULL`.
  - Example: Return all employees who are not in a department.
    ```sql
    SELECT ename
    FROM emp
    WHERE dno IS NULL
    ```
  - Example: Return all departments that have a manager.
    ```sql
    SELECT dname
    FROM dept
    WHERE mgreno IS NOT NULL
    ```

**Set Operations**

- The set operations of union, intersection, and difference are used to combine the results of two SQL queries.
  - `UNION`, `INTERSECT`, `EXCEPT`  
  - Note: `UNION ALL` returns all rows
  - Example: Return the employees who are either directly supervised by 'R. Davis' or directly supervised by 'M. Smith'.
    ```sql
    (SELECT E.ename
    FROM emp as E, emp as M
    WHERE E.supereno = M.eno and M.ename='R. Davis')
    UNION
    (SELECT E.ename
    FROM emp as E, emp as M
    WHERE E.supereno = M.eno and M.ename='M. Smith')
    ```

**Ordering Result Data**

- The query result returned is not ordered on any attribute by default. We can order the data using the `ORDER BY` clause:
  ```sql
  SELECT ename, salary, bdate
  FROM emp
  WHERE salary > 30000
  ORDER BY salary DESC, ename ASC
  ```
  - 'ASC' sorts the data in ascending order, and 'DESC' sorts it in descending order. The default is 'ASC'.
  - The order of sorted attributes is significant. The first attribute specified is sorted on first, then the second attribute is used to break any ties, etc.
  - NULL is normally treated as less than all non-null values.
SQL Querying with NULL and LIKE

- **Question:** What query would return the department names that do not have a manager or contain 'ent'.

  A) SELECT dname
     FROM dept
     WHERE mgreno = NULL OR dname LIKE '_ent'

  B) SELECT dname
     FROM dept
     WHERE mgreno IS NULL OR dname LIKE '%ent%'

SQL...What we have seen thus far

- "basic" SQL
  - basic SELECT statement
  - JOINS
  - SET operations
  - Tuple variables

- Next: Advanced SQL
  - Mix in Relational calculus operators
  - Add in features
  - SET membership operations
  - IN, <ALL, CONTAINS, etc.
  - Nested Queries
  - Tuple Variables
  - Aggregate Operators

SQL—Aggregate Operations

- Thus far SQL (and Relational Algebra/Calculus) only fetched data stored in database tables
- What if we need some basic 'statistics' on the data?
  - Number of rows?
  - Maximum value in a field?
- Aggregate Operators: apply a function to a set of tuples
  - Function defined on one (or more) field
- Aggregate operators are extension to Relational algebra

Aggregate Queries and Functions

- Several queries cannot be answered using the simple form of the SELECT statement. These queries require a summary calculation to be performed. Examples:
  - What is the maximum employee salary?
  - What is the total number of hours worked on a project?
  - How many employees are there in department 'D1'?

- To answer these queries requires the use of aggregate functions. These functions operate on a single column of a table and return a single value.
Aggregate Operators

- Compute functions on set of tuples selected by where clause
- Semantics: if SELECT clause contains aggregate operations then it can contain only aggregate operations
  - Except when groupby construct is used
  - Functions on sets of values but result is single value
  - Average, minimum, maximum, sum, count(size)

Aggregate Functions

- The five basic aggregate functions are:
  - COUNT - returns the # of values in a column
  - SUM - returns the sum of the values in a column
  - AVG - returns the average of the values in a column
  - MIN - returns the smallest value in a column
  - MAX - returns the largest value in a column

- Notes:
  - 1) COUNT, MAX, and MIN apply to all types of fields, whereas SUM and AVG apply to only numeric fields.
  - 2) Except for COUNT(*) all functions ignore nulls. COUNT(*) returns the number of rows in the table.
  - 3) Use DISTINCT to eliminate duplicates.

Aggregate Function Example

- Return the number of employees and their average salary.

```
SELECT COUNT(eno) AS numEmp, AVG(salary) AS avgSalary
FROM emp
```

Result

<table>
<thead>
<tr>
<th>numEmp</th>
<th>avgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>38750</td>
</tr>
</tbody>
</table>

Aggregate Operators

- Significant extension of relational algebra.

```
SELECT AVG(balance)
FROM Deposit;
```

```
SELECT D.CustID
FROM Deposit D
WHERE D.balance = (SELECT MAX(S.balance)
                    FROM Deposit S);
```

```
SELECT COUNT(DISTINCT CustID)
FROM Deposit;
```

```
SELECT AVG (D.balance)
FROM Deposit D
WHERE D.branchname='GW';
```

```
SELECT COUNT (DISTINCT CustID)
FROM Deposit
```
More Examples

Purchase(product, date, price, quantity)

```sql
SELECT Sum(price * quantity) 
FROM Purchase
```

What do they mean?

```sql
SELECT Sum(price * quantity) 
FROM Purchase
WHERE product = 'bagel'
```

Simple Aggregations

Purchase

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

```sql
SELECT Sum(price * quantity) 
FROM Purchase
WHERE product = 'bagel'
```

50 (= 20+30)

Motivation for Grouping

- 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 average balance for each branch in the bank.
  - In general, we don’t know how many branches exist, and what the balances are!
  - Suppose we know that 10 branchnames exists; then we can write 10 queries that look like this (!):

```
SELECT AVG(balance) 
FROM Deposit D 
WHERE D.branchname='x'
```

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

GROUP BY Clause

- Aggregate functions are most useful when combined with the GROUP BY clause. The GROUP BY clause groups the tuples based on the values of the attributes specified.
- When used in combination with aggregate functions, the result is a table where each tuple consists of unique values for the group by attributes and the result of the aggregate functions applied to the tuples of that group.
Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

Let's see what this means…

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause: grouped attributes and aggregates.

3. SELECT

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>

PRODUCT | DATE | PRICE | QUANTITY
--------|------|-------|----------
Bagel   | 10/21| 1     | 20       |
Bagel   | 10/25| 1.50  | 20       |
Banana  | 10/3 | 0.5   | 10       |
Banana  | 10/10| 1     | 10       |
GROUP BY Example

- For each employee title, return the number of employees with that title, and the minimum, maximum, and average salary.

```sql
SELECT title, COUNT(eno) AS numEmp,
       MIN(salary) as minSal,
       MAX(salary) as maxSal,
       AVG(salary) AS avgSal
FROM emp
GROUP BY title
```

<table>
<thead>
<tr>
<th>title</th>
<th>numEmp</th>
<th>minSal</th>
<th>maxSal</th>
<th>avgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td>2</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
</tr>
<tr>
<td>SA</td>
<td>3</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>ME</td>
<td>2</td>
<td>40000</td>
<td>40000</td>
<td>40000</td>
</tr>
<tr>
<td>PR</td>
<td>1</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
</tbody>
</table>

GROUP BY Clause Rules

- There are a few rules for using the `GROUP BY` clause:
  1) A column name cannot appear in the `SELECT` part of the query unless it is part of an aggregate function or in the list of group by attributes.
     Note that the reverse is allowed: a column can be in the `GROUP BY` without being in the `SELECT` part.
  2) Any `WHERE` conditions are applied before the `GROUP BY` and aggregate functions are calculated.

Find average balance at each branch

```sql
SELECT branchname, avg(balance)
FROM Deposit
GROUP BY branchname;
```

Condition on the Groups

- What if we are only interested in groups that satisfy a condition?
**HAVING Clause**

- The **HAVING** clause is applied **AFTER** the **GROUP BY** clause and aggregate functions are calculated.
- It is used to filter out entire groups that do not match certain criteria.
- The **HAVING** clause can contain any condition that references aggregate functions and the group by attributes themselves.
  - However, any conditions on the **GROUP BY** attributes should be specified in the **WHERE** clause if possible due to performance reasons.

**Find branches where average account balance is greater than $1200**

SELECT branchname, avg(balance)
FROM Deposit
GROUP BY branchname
HAVING avg(balance) > 1200

Only the group (branchnames) satisfying the HAVING clause is selected

**Grouping and Aggregation: Evaluation Steps**

**Purchase(product, date, price, quantity)**

Find total sales after 10/1/2005 per product.

SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product

Let's see what this means...

**Find branches where average account balance is greater than $1200**

SELECT branchname, avg(balance)
FROM Deposit
GROUP BY branchname
HAVING avg(balance) > 1200

Only the group (branchnames) satisfying the HAVING clause is selected

**Grouping and Aggregation**

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>

What if we are only interested in products that sold quantity >30?

SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING Clause

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product, except that we consider only products that had at least 30 buyers.

```
SELECT product, Sum(price * quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.

General form of Grouping and Aggregation

```
SELECT S
FROM R1,…,Rn
WHERE C1
GROUP BY a1,…,ak
HAVING C2
```

S = may contain attributes a1,…,ak and/or any aggregates but NO OTHER ATTRIBUTES
C1 = is any condition on the attributes in R1,…,Rn
C2 = is any condition on aggregate expressions

Why?

Generalized SELECT: Queries With GROUP BY and HAVING

```
SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE qualification/predicate
GROUP BY grouping-list
HAVING group-qualification/predicate
```

- The attribute-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (balance)).
- The attribute list 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 relation-list is computed, tuples that fail qualification in WHERE clause 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 specified in the HAVING clause is then applied to eliminate some groups. Expressions in HAVING clause 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.
- Any aggregate function can be applied to a group
  - Final SELECT can have function over each selected group

**HAVING Example**

- Return the title and number of employees of that title where the number of employees with the title is at least 2.

```sql
SELECT title, COUNT(eno) AS numEmp
FROM emp
GROUP BY title
HAVING COUNT(eno) >= 2
```

<table>
<thead>
<tr>
<th>title</th>
<th>numEmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td>2</td>
</tr>
<tr>
<td>SA</td>
<td>3</td>
</tr>
<tr>
<td>ME</td>
<td>2</td>
</tr>
</tbody>
</table>

**GROUP BY/HAVING Example**

- For employees born after December 1, 1965, return the average salary by department where the average is > 40,000.

```sql
SELECT dname, AVG(salary) AS avgSal
FROM emp, dept
WHERE emp.dno = dept.dno and emp.bdate > DATE '1965-12-01'
GROUP BY dname
HAVING AVG(salary) > 40000
```

**Step #1: Perform Join and Filter in WHERE clause**

```
eno ename bdate title salary supereno dno dname mreno
E2 M. Smith 1966-06-04 SA 50000 E3 D3 Accounting E5
E3 A. Lee 1966-07-05 ME 50000 E7 D2 Consulting E7
E5 B. Casey 1971-12-23 SA 50000 E8 D3 Accounting E3
E7 R. Davis 1977-09-08 ME 40000 E8 D1 Management E8
E8 J. Jones 1972-10-11 SA 50000 null D1 Management E8
```
GROUP BY/HAVING Example (2)

Step #2: GROUP BY on dname

<table>
<thead>
<tr>
<th>eno</th>
<th>ename</th>
<th>bdate</th>
<th>title</th>
<th>salary</th>
<th>supereno</th>
<th>dno</th>
<th>dname</th>
<th>mgreno</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2</td>
<td>M. Smith</td>
<td>1966-06-04</td>
<td>SA</td>
<td>50000</td>
<td>E5</td>
<td>D3</td>
<td>Accounting</td>
<td>E5</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
<td>1971-12-25</td>
<td>SA</td>
<td>50000</td>
<td>E8</td>
<td>D3</td>
<td>Accounting</td>
<td>E5</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
<td>1966-06-04</td>
<td>ME</td>
<td>40000</td>
<td>E7</td>
<td>D2</td>
<td>Consulting</td>
<td>E5</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
<td>1977-09-08</td>
<td>ME</td>
<td>40000</td>
<td>E8</td>
<td>D1</td>
<td>Management</td>
<td>E8</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
<td>1972-10-11</td>
<td>SA</td>
<td>null</td>
<td>D1</td>
<td></td>
<td>Management</td>
<td>E8</td>
</tr>
</tbody>
</table>

Step #3: Calculate aggregate functions

<table>
<thead>
<tr>
<th>dname</th>
<th>avgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>50000</td>
</tr>
<tr>
<td>Consulting</td>
<td>40000</td>
</tr>
<tr>
<td>Management</td>
<td>45000</td>
</tr>
</tbody>
</table>

Step #4: Filter groups using HAVING clause

<table>
<thead>
<tr>
<th>dname</th>
<th>avgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>50000</td>
</tr>
<tr>
<td>Management</td>
<td>45000</td>
</tr>
</tbody>
</table>

GROUP BY/HAVING

Multi-Attribute Example

- Return the employee number, department number and hours the employee worked per department where the hours is >= 10.

```sql
SELECT eno, dno, SUM(hours)
FROM workson AS W, dept AS D, proj AS P
WHERE W.pno = P.pno and P.dno = D.dno
GROUP BY W.eno, D.dno
HAVING SUM(hours) >= 10
```

Result:

<table>
<thead>
<tr>
<th>eno</th>
<th>dno</th>
<th>SUM(hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>D1</td>
<td>12</td>
</tr>
<tr>
<td>E2</td>
<td>D1</td>
<td>24</td>
</tr>
<tr>
<td>E3</td>
<td>D2</td>
<td>48</td>
</tr>
<tr>
<td>E4</td>
<td>D2</td>
<td>18</td>
</tr>
<tr>
<td>E5</td>
<td>D2</td>
<td>24</td>
</tr>
<tr>
<td>E6</td>
<td>D2</td>
<td>48</td>
</tr>
<tr>
<td>E7</td>
<td>D3</td>
<td>36</td>
</tr>
</tbody>
</table>

Question:
1) How would you only return records for departments D2 and D3?

Subqueries

- SQL allows a single query to have multiple subqueries nested inside of it. This allows for more complex queries to be written.

- When queries are nested, the outer statement determines the contents of the final result, while the inner SELECT statements are used by the outer statement (often to lookup values for WHERE clauses).

- A subquery can be in the SELECT, FROM, WHERE or HAVING clause.

```sql
SELECT ename, salary, bdate
FROM emp
WHERE salary > (SELECT AVG(salary) FROM emp)
```

Nested Queries: Semantics

- Need to be careful about scope of tuple variables

  Scoping rules: local definition and then global

  In subquery – legal to use only tuple variables defined in subquery itself or in any query that contains the subquery

- Evaluate subquery at each reference

  - Construct cross product of tables in FROM clause
  - For each row when testing predicate conditions in WHERE clause
  - Recompute subquery
    - Is this really necessary?
  - If subquery contains another subquery then apply same principle
Types of Subqueries

- There are three types of subqueries:
  1) **scalar subquery** - return a single value. Often value is then used in a comparison.
     - If query is written so that it expects a subquery to return a single value, and if it returns multiple values or no values, a run-time error occurs.
  2) **row subquery** - returns a single row which may have multiple columns.
  3) **table subquery** - returns one or more columns and multiple rows.

Scalar Subquery Examples

Return the employees that are in the ‘Accounting’ department:

```
SELECT ename
FROM emp
WHERE dno = (SELECT dno
             FROM dept
             WHERE dname = 'Accounting')
```

Return all employees who work more hours than average on a single project:

```
SELECT ename
FROM emp, workson
WHERE workson.eno = emp.eno AND
workson.hours > (SELECT AVG(hours)
                 FROM workson)
```

Table Subqueries and Set membership Operators

- A table subquery returns a relation. There are several operators on sets that can be used to test condition on subquery result:
  - **EXISTS** $R$ - true if $R$ is not empty
  - $s$ **IN** $R$ - true if $s$ is equal to one of the values of $R$
  - $s$ **> ALL** $R$ - true if $s$ is greater than every value in $R$
  - $s$ **> ANY** $R$ - true if $s$ is greater than any value in $R$

- Notes:
  - 1) Any of the comparison operators (<, <=, =, etc.) can be used.
  - 2) The keyword **NOT** can proceed any of the operators.
    - Example: $s$ **NOT IN** $R$

Set Membership Operations: summary

- Can check for set membership using **IN** and **NOT IN**
  - **x IN A** or **x NOT IN A**
  - Implements Relational Calculus operators
  - **IN** connective tests for membership in the set $A$
  - **NOT IN** tests for absence of tuples
  - Can test using multiple attribute element

- Set existence using **EXISTS**
  - Returns true if the argument subquery is nonempty (the converse for the NOT EXIST) thus checking for empty relations
**Subqueries Returning Relations and Set Membership operators**

Company(name, city)  
Product(pname, maker)  
Purchase(id, product, buyer)  

Return cities of companies that manufacture products bought by Joe Blow

```sql
SELECT Company.city  
FROM Company  
WHERE Company.name IN  
  (Set of Companies that manufacture products bought by Joe Blow);  
/* write a SELECT query to obtain this set */
```

**Subquery...**

Set of Companies that manufacture products bought by Joe Blow

```sql
(SELECT Product.maker  
FROM Purchase , Product  
WHERE Product.pname = Purchase.product  
AND Purchase .buyer = 'Joe Blow')
```

**Subqueries Returning Relations**

Company(name, city)  
Product(pname, maker)  
Purchase(id, product, buyer)  

Return cities of companies that manufacture products bought by Joe Blow

```sql
SELECT Company.city  
FROM Company  
WHERE Company.name IN  
  (SELECT Product.maker  
   FROM Purchase , Product  
   WHERE Product.pname = Purchase.product  
   AND Purchase .buyer = 'Joe Blow');
```

**Table Subquery Examples**

Return all departments who have a project with a budget greater than $300,000:

```sql
SELECT dname FROM dept  
WHERE dno IN  
  (SELECT dno FROM proj  
   WHERE budget > 300000)
```

Return all projects that 'J. Doe' works on:

```sql
SELECT pname FROM proj  
WHERE pno IN  
  (SELECT pno FROM workson  
   WHERE eno =  
     (SELECT eno FROM emp  
      WHERE ename = 'J. Doe'))
```
Set Membership: Quantifiers

Product (pname, price, company)
Company (cname, city)

Find all companies that make some products with price < 100

Existential: easy ! 😊

Set Membership: Quantifiers

Product (pname, price, company)
Company (cname, city)

Find all companies that make some products with price < 100

SELECT DISTINCT Company.cname
FROM Company, Product
WHERE Company.cname = Product.company and Product.price < 100

Existential: easy ! 😊

Set Membership: Quantifiers

Product (pname, price, company)
Company (cname, city)

Find all companies that make only products with price < 100

same as:

Find all companies such that all of their products have price < 100

Recall equivalence: Forall x P(x) = Not Exists x (Not P(x))

Universal: hard ! 😊

Set Membership: Quantifiers

1. Find the other companies: i.e. s.t. some product >= 100

SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
FROM Product
WHERE Product.price >= 100)
Set Membership: Quantifiers

1. Find the other companies: i.e. s.t. some product \( \geq 100 \)
   
   ```sql
   SELECT DISTINCT Company.cname
   FROM Company
   WHERE Company.cname IN (SELECT Product.company
   FROM Product
   WHERE Product.price >= 100)
   ```

2. Find all companies s.t. all their products have price < 100
   
   ```sql
   SELECT DISTINCT Company.cname
   FROM Company
   WHERE Company.cname NOT IN (SELECT Product.company
   FROM Product
   WHERE Product.price >= 100)
   ```

EXISTS Example

- The EXISTS function is used to check whether the result of a nested query is empty or not.
  - EXISTS returns true if the nested query has 1 or more tuples.

- Example: Return all employees who have the same name as someone else in the company.
  
  ```sql
  SELECT ename
  FROM emp
  WHERE EXISTS (SELECT *
                 FROM emp AS E2
                 WHERE E.ename = E2.ename AND
                 E.eno <> E2.eno)
  ```

ANY and ALL Example

- ANY means that any value returned by the subquery can satisfy the condition.
- ALL means that all values returned by the subquery must satisfy the condition.

- Example: Return the employees who make more than all the employees with title 'ME' make.
  
  ```sql
  SELECT ename
  FROM emp AS E
  WHERE salary > ALL (SELECT salary
                      FROM emp
                      WHERE title = 'ME')
  ```

Subquery Syntax Rules

- 1) The ORDER BY clause may not be used in a subquery.

- 2) The number of attributes in the SELECT clause in the subquery must match the number of attributes compared to with the comparison operator.

- 3) Column names in a subquery refer to the table name in the FROM clause of the subquery by default. You must use aliasing if you want to access a table that is present in both the inner and outer queries.
More Set Membership Operations

- Previous operators allowed checking for existence
- SQL provides operators to test elements of one set \( A \) with elements on another set \( B \)
  - **SOME**: \( op \) SOME
    - Also called as **ANY** in some versions
  - **ALL**: \( op \) ALL
    - \( op \) can be \( \geq, >, <, <\geq, =, \neq \)
- Test single value against members of an entire set
  - \( x \geq \) \( ALL \) \((R)\)

Find branches that have greater assets than some branch located in Brooklyn

- \( A = \) Set of branches in Brooklyn
- \( \text{Branch.assets} > \text{some} \)
- Compare assets of branch to members of set \( A \)
  - Branch is selected if its assets is greater than assets of SOME branch in Brooklyn

\[
\text{SELECT branchname}
\text{FROM Branch}
\text{WHERE assets} > \text{(SELECT assets FROM Branch WHERE branchcity='Brooklyn');}
\]
- Inner select has set of assets of branches in Brooklyn
- For branches that have assets greater than all branches in Brooklyn replace >some with >all

Set Comparison Operations: subset

- Check if one set (query result) contains another set (query result)
  - Is \( A \) subset of \( B \)?
  - Is \( A \) not a subset of \( B \)?
- **contains** and **not contains** operators
**Set Membership Examples:**

Find all customers who have an account at all branches located in Brooklyn

SELECT CustID
FROM Deposit S
WHERE
  (set of branchnames where customer has an account)
  contains
  (set of all branchnames located in Brooklyn)

**Set existence example**

- Test if subquery is empty
  - Exists returns true if argument is nonempty
- Find all customers who have both an account and a loan at Downtown branch
  - First test if customer has account and second test if customer has loan
    - Exists in Deposit and Exists in Loan

SELECT C.CustID
FROM Customer C
WHERE exists (SELECT *
  FROM Deposit D
  WHERE D.CustID=C.CustID
  AND D.branchname='Downtown')
AND exists (SELECT *
  FROM Loan L
  WHERE L.CustID= C. CustID
  AND L.branchname='Downtown');
Correlated Subqueries
- Most queries involving subqueries can be rewritten so that a subquery is not needed.
  - This is normally beneficial because query optimizers may not do a good job at optimizing queries containing subqueries.
- A nested query is **correlated** with the outside query if it must be re-computed for every tuple produced by the outside query. Otherwise, it is **uncorrelated**, and the nested query can be converted to a non-nested query using joins.
- A nested query is correlated with the outer query if it contains a reference to an attribute in the outer query.

Correlated Subquery Example
Return all employees who have the same name as another employee:

```sql
SELECT ename
FROM emp as E
WHERE EXISTS (SELECT eno FROM emp as E2
WHERE E.ename = E2.ename AND E.eno <> E2.eno)
```

A more efficient solution with joins:

```sql
SELECT E.ename
FROM emp as E, emp as E2
WHERE E.ename = E2.ename AND E.eno <> E2.eno
```

Subqueries in FROM Clause
- Subqueries are used in the FROM clause to produce temporary table results for use in the current query.
- Example: Return the departments that have an employee that makes more than $40,000.

```sql
SELECT dname
FROM Dept D, (SELECT ename, dno FROM Emp
WHERE salary > 40000) E
WHERE D.dno = E.dno
```

Subqueries and GroupBy/Aggregate Operations
Question: Find those branches with the highest average balance.

- Aggregate functions cannot be composed
  - Max(min(...)) not allowed
- Select groups that have average balance greater than or equal to ALL average balances at all branches.
  - Need nested query to compute set of balances at each branch
SELECT branchname
FROM Deposit
GROUP BY branchname
HAVING AVG(balance) >= ALL
(SELECT AVG(balance)
FROM Deposit
GROUP BY branchname);

SELECT AVG(balance)
FROM Deposit D, Customer C
WHERE D.CustID=C.CustID AND C.City='New York'
GROUP BY D.CustID
HAVING COUNT(distinct account-number)>=3

Find average balance of all depositors (account holders) who live in New York and have at least three accounts
- Select tuples where Customer lives in New York
- Form groups in Deposit based on Customer ID
  - GROUP BY CustID
- Select only those groups with 3 or more tuples
  - COUNT number of tuples in each group

A quick Note:
Group-by v.s. Nested Query

Author(login,name)
Wrote(login,url)

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

SELECT DISTINCT Author.name
FROM Author
WHERE count(SELECT Wrote.url
FROM Wrote
WHERE Author.login=Wrote.login)
> 10

This is SQL by a novice
Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT Author.name
FROM   Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.name
HAVING count(wrote.url) > 10
```

No need for DISTINCT: automatically from GROUP BY

Next...

- Database update operations
  - Insert, Delete...
- Views
- Constraints