SQL:
Data Manipulation Language

Part 2

SQL Arithmetic Operators

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder

From http://www.tutorialspoint.com/sql/sql-operators.htm

SQL Comparison Operators	
Operator	**Description**
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

From http://www.tutorialspoint.com/sql/sql-operators.htm

SQL Logical Operators

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE*	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). *Not

From http://www.tutorialspoint.com/sql/sql-operators.htm

Logic

- Two-valued logic, aka 2VL (George Boole)
  - True
  - False
- SQL uses 3VL (Jan Lukasiewicz)
  - True
  - Unknown
  - False

2VL Truth Tables

AND	True	False
True	True	False
False	False	False

OR	True	False
True	True	True
False	True	False

NOT	True	False
	False	True

3VL Truth Tables

AND	True	Unk	False
True	True	Unk	False
Unk	Unk	Unk	False
False	False	False	False

OR	True	Unk	False
True	True	True	True
Unk	True	Unk	Unk
False	True	Unk	False

NOT	True	Unk	False
	False	Unk	True

NULL

NOTE:

The following queries were run in PostgreSQL 9.5.

This syntax will not work in MySQL.

Additionally, the following psql shell command was used to change how NULLs are displayed: `\pset null '[NULL]'`

NULL

- `SELECT true AND NULL;`
- `SELECT false AND NULL;`

- `SELECT true OR NULL;`
- `SELECT false OR NULL;`

NULL

- `SELECT true AND NULL;`

?column?
[NULL]

- `SELECT false AND NULL;`

- `SELECT true OR NULL;`

- `SELECT false OR NULL;`

NULL

- `SELECT true AND NULL;`

?column?
[NULL]

- `SELECT false AND NULL;`

?column?
f

- `SELECT true OR NULL;`

- `SELECT false OR NULL;`

NULL

- `SELECT true AND NULL;`

?column?
[NULL]

- `SELECT false AND NULL;`

?column?
f

- `SELECT true OR NULL;`

?column?
t

- `SELECT false OR NULL;`

NULL

- `SELECT true AND NULL;`

?column?
[NULL]

- `SELECT false AND NULL;`

?column?
f

- `SELECT true OR NULL;`

?column?
t

- `SELECT false OR NULL;`

?column?
[NULL]

NULL

- `SELECT NOT NULL;`

NULL

- `SELECT NOT NULL;`

?column?
[NULL]

NULL

- `SELECT 1 WHERE 1 = 1;`
- `SELECT 1 WHERE 1 = 0;`
- `SELECT 1 WHERE 1 = NULL;`
- `SELECT 1 WHERE NULL = NULL;`

The WHERE will return a row only if the condition evaluates to true.
UNKNOWN isn't true

NULL

- `SELECT 1 WHERE 1 = 1;`

?column?
1

- `SELECT 1 WHERE 1 = 0;`

- `SELECT 1 WHERE 1 = NULL;`

- `SELECT 1 WHERE NULL = NULL;`

The WHERE will return a row only if the condition evaluates to true. UNKNOWN isn't true

NULL

- `SELECT 1 WHERE 1 = 1;`

?column?
1

- `SELECT 1 WHERE 1 = 0;`

?column?

- `SELECT 1 WHERE 1 = NULL;`

- `SELECT 1 WHERE NULL = NULL;`

The WHERE will return a row only if the condition evaluates to true. UNKNOWN isn't true.

NULL

- `SELECT 1 WHERE 1 = 1;`

```
?column?
       1
```

- `SELECT 1 WHERE 1 = NULL;`

```
?column?
```

- `SELECT 1 WHERE 1 = 0;`

```
?column?
```

- `SELECT 1 WHERE NULL = NULL;`

The WHERE will return a row only if the condition evaluates to true. UNKNOWN isn't true

NULL

- `SELECT 1 WHERE 1 = 1;`

?column?
1

- `SELECT 1 WHERE 1 = NULL;`

?column?

- `SELECT 1 WHERE 1 = 0;`

?column?

- `SELECT 1 WHERE NULL = NULL;`

?column?

The WHERE will return a row only if the condition evaluates to true. UNKNOWN isn't true.

NULL

- `SELECT NULL = NULL;`

- `SELECT NULL != NULL;`

NULL

- `SELECT NULL = NULL;`

?column?
[NULL]

- `SELECT NULL != NULL;`

NULL

- `SELECT NULL = NULL;`

?column?
[NULL]

- `SELECT NULL != NULL;`

?column?
[NULL]

NULL

How do we test for NULL?

- `SELECT NULL = NULL;`

?column?
[NULL]

- `SELECT NULL != NULL;`

?column?
[NULL]

NULL

The IS NULL operator .

- `SELECT 1 IS NULL;`

?column?
f

- `SELECT 1 IS NOT NULL;`

?column?
t

- `SELECT NULL IS NULL;`

?column?
t

- `SELECT NULL IS NOT NULL;`

?column?
f

NULL

- `SELECT 1 + NULL;`

?column?
[NULL]

- `SELECT 1 < NULL;`

?column?
[NULL]

- `SELECT 1 > NULL;`

?column?
[NULL]

- `SELECT 1 <> NULL;`

?column?
[NULL]

NULL

- `SELECT 0 BETWEEN 0 AND 2;`

?column?
t

- `SELECT 1 BETWEEN 0 AND 2;`

?column?
t

- `SELECT 0 BETWEEN 0 AND -1;`

?column?
f

NULL

- `SELECT 0 BETWEEN 0 AND NULL;`

NULL

- `SELECT 0 BETWEEN 0 AND NULL;`

?column?
[NULL]

IN vs EXISTS vs JOIN

- **IN**:
  - Returns true if a specified value matches any value in a subquery or a list.
- **EXIST**:
  - Returns true if a subquery contains any rows.
- **JOIN**:
  - Joins 2 result sets on the joining

NULL

- `SELECT EXISTS(SELECT NULL);`
- `SELECT NOT EXISTS(SELECT NULL);`

NULL

- `SELECT EXISTS(SELECT NULL);`

?column?
t

- `SELECT NOT EXISTS(SELECT NULL);`

NULL

- `SELECT EXISTS(SELECT NULL);`

?column?
t

- `SELECT NOT EXISTS(SELECT NULL);`

?column?
f

NULL

- `SELECT 1 IN (1);`
- `SELECT 1 NOT IN (1);`
- `SELECT 1 IN (NULL);`
- `SELECT 1 NOT IN (NULL);`

NULL

- `SELECT 1 IN (1);`

?column?
t

- `SELECT 1 NOT IN (1);`

- `SELECT 1 IN (NULL);`

- `SELECT 1 NOT IN (NULL);`

NULL

- `SELECT 1 IN (1);`

?column?
t

- `SELECT 1 IN (NULL);`

- `SELECT 1 NOT IN (1);`

?column?
f

- `SELECT 1 NOT IN (NULL);`

NULL

- `SELECT 1 IN (1);`

?column?
t

- `SELECT 1 IN (NULL);`

?column?
[NULL]

- `SELECT 1 NOT IN (1);`

?column?
f

- `SELECT 1 NOT IN (NULL);`

NULL

- `SELECT 1 IN (1);`

?column?
t

- `SELECT 1 IN (NULL);`

?column?
[NULL]

- `SELECT 1 NOT IN (1);`

?column?
f

- `SELECT 1 NOT IN (NULL);`

?column?
[NULL]

NULL

```
SELECT *
FROM (
    VALUES (NULL), (NULL)
) AS T;
```

NULL

```
SELECT *
FROM (
    VALUES (NULL), (NULL)
) AS T;
```

column1
[NULL]
[NULL]

NULL

```
SELECT DISTINCT *
FROM (
    VALUES (NULL), (NULL)
) AS T;
```

NULL

```
SELECT DISTINCT *
FROM (
    VALUES (NULL), (NULL)
) AS T;
```

column1
[NULL]

NULL

```
DROP TABLE IF EXISTS T1;
DROP TABLE IF EXISTS T2;

CREATE TABLE t1 (
  a INT
);


INSERT INTO t1 (a)
  VALUES (1), (2),(null);


CREATE TABLE t2 AS
  SELECT * FROM t1;
```

t1
a
1
2
null

t2
a
1
2
null

NULL

```
SELECT *
FROM t1, t2
WHERE t1.a = t2.a;
```

t1
a
1
2
null

t2
a
1
2
null

NULL

```
SELECT *
FROM t1, t2
WHERE t1.a = t2.a;
```

t1
a
1
2
null

t2
a
1
2
null

a	a
1	1
2	2

NULL

```
SELECT *
FROM t1 NATURAL JOIN  t2;
```

t1
a
1
2
null

t2
a
1
2
null

NULL

```
SELECT *
FROM t1 NATURAL JOIN  t2;
```

t1
a
1
2
null

t2
a
1
2
null

a	a
1	1
2	2

NULL

```
SELECT * FROM t1
UNION ALL
SELECT * FROM t2;
```

t1
a
1
2
null

t2
a
1
2
null

NULL

```
SELECT * FROM t1
UNION ALL
SELECT * FROM t2;
```

a
1
2
null
1
2
null

t1
a
1
2
null

t2
a
1
2
null

NULL

```
SELECT * FROM t1
UNION
SELECT * FROM t2;
```

t1
a
1
2
null

t2
a
1
2
null

NULL

t1
a
1
2
null

t2
a
1
2
null

```
SELECT * FROM t1
UNION
SELECT * FROM t2;
```

a
1
2
null

NULL

More on NULLs:

- http://www.xaprb.com/blog/2006/05/18/why-null-never-compares-false-to-anything-in-sql/

Advanced Subqueries

- FROM (*subquery*)
- WHERE *column_name* < (*subquery*)
  - Can replace < with >, =, <=, >=, or <>
- SELECT (*subquery*)

Aggregation

- Aggregate functions take a collection of values as input and return a single value.
- SQL offers five built-in aggregate functions:
  - Average: **AVG**
  - Minimum: **MIN**
  - Maximum: **MAX**
  - Total: **SUM**
  - Count: **COUNT**

From Silbershatz

Aggregation

- Aggregate functions take a collection of values as input and return a single value.
- SQL offers five built-in aggregate functions:
  - Average: **AVG**
  - Minimum: **MIN**
  - Maximum: **MAX**
  - Total: **SUM**
  - Count: **COUNT**

The input to SUM and AVG must be collections of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

Except COUNT, all aggregate operations apply to a single attribute.

From Silbershatz

Aggregation

- Retaining duplicates is important in computing AVGs, SUMs, and COUNTs.

- There are cases where we must eliminate duplicates before computing an aggregate function.
    - If we want to eliminate duplicates, we use the keyword DISTINCT in the aggregate expression.

From Silbershatz

Aggregation

```
SELECT SUM(a)
FROM (VALUES (1), (1)) AS
T(a);
```

sum
2

```
SELECT SUM(DISTINCT a)
FROM (VALUES (1), (1)) AS
T(a);
```

sum
1

Query 32

Calculate the average salary of all employees.

Note: Averages non-null salary values.

Query 32

Calculate the average salary of all employees.

```
SELECT AVG(fsalary)
FROM Faculty;
```

Note: Averages non-null salary values.

Query 33

Display Faculty rows that earn more than the average salary.

Query 33

Display Faculty rows that earn more than the average salary.

```
SELECT *
FROM Faculty
   WHERE fsalary >
        (SELECT AVG(fsalary)
            FROM Faculty);
```

Query 34

Find departments with faculty salaries over their budgeted amount.

Query 34

Find departments with faculty salaries over their budgeted amount.

```
SELECT *
FROM Department
WHERE dsalary_budget <
  (SELECT SUM(fsalary)
     FROM Faculty
    WHERE ddept = fdept);
```

Aggregation with Grouping

- To apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples, use the **GROUP BY** clause.
    - The attribute or attributes given in the GROUP BY clause are used to form groups.
    - Tuples with the same value on all attributes in the GROUP BY clause are placed in one group.

From Silbershatz

GROUP BY Syntax Issues

- All columns SELECTed
    - must be in GROUP BY
      or the target of an aggregate function.

Query 35

Find the highest salary in each department.

Query 35

Find the highest salary in each department.

```
SELECT fdept, MAX(fsalary)
FROM Faculty
GROUP BY fdept;
```

GROUP BY Syntax Issues

- When a SQL query uses grouping, it is important to ensure that *the only attributes that appear in the* SELECT *statement without being aggregated are those that appear in the* GROUP BY *clause.*

Silbershatz

GROUP BY Syntax Issues

- Any attribute that is not present in the GROUP BY clause must appear only inside an aggregate function if it appears in the SELECT clause.

Silbershatz

The HAVING Clause

- At times, if is useful to state a condition that applies to groups rather than to tuples.
- SQL applies predicates in the HAVING clause after groups have been formed, so aggregate functions may be used.
- Any attribute that is present in the HAVING clause without being aggregated must appear in the GROUP BY clause.

Silbershatz

SQL Seduction 7

Find the highest salary in each department and include fid in the output.

SQL Seduction 7

Find the highest salary in each department and include fid in the output.

```
SELECT fid, fdept, MAX(fsalary)
FROM Faculty
GROUP BY fdept;
```

SQL Seduction 7

Find the highest salary in each department and include fid in the output.

~~SELECT fid, fdept, MAX(fsalary)~~
~~FROM Faculty~~
~~GROUP BY fdept;~~

Syntax error! fid not in GROUP BY!

Query 36

This fixes the syntax error, but it doesn’t solve the original problem.

```
SELECT fid, fdept, MAX(fsalary)
FROM Faculty
GROUP BY fdept, fid;
```

WRONG result!

WHY?

Query 37

This solves the original problem.

```
SELECT fid, fdept, fsalary
FROM (SELECT fdept AS mdept, MAX(fsalary) AS msalary
      FROM Faculty
      GROUP BY fdept) AS t,
     Faculty
WHERE (fdept = mdept)
AND fsalary = msalary;
```

Warning: The order of the “tables” matters in MySQL.

Query 38

List all departments with only one course.

Query 38

List all departments with only one course.

```
SELECT cdept FROM Course
GROUP BY cdept
HAVING count(*) = 1;
```

HAVING is to selecting groups as WHERE is to selecting rows.

Aggregate Function Issues

- Can be SELECTed
  - Typically used with GROUP BY
- Can appear in HAVING clause
-
- Never allowed as a WHERE clause's simple condition, such as ~~WHERE COUNT(*) = 1~~

Sequence of Operations

- **WHERE** chooses rows
- **GROUP BY** groups chosen rows
- **HAVING** chooses groups
- **ORDER BY** sequences result
- **SELECT** chooses columns to display
- **DISTINCT** compresses duplicate result rows

String Operations

- SQL specifies strings by enclosing them in single quotes.
  - A single quote character that is part of a string can be specified by using two single quote characters: 'Who"s paying attention?'

String Operations

- The SQL standard specifies that equality operations on strings is case sensitive, however some databases do not distinguish uppercase and lowercase while comparing strings.

MySQL:

```
SELECT  'foo' = 'FOO';
```

'foo' = 'FOO'
1

PostgreSQL:

```
SELECT  'foo' = 'FOO';
```

?column?
f

SQLite:

```
SELECT  'foo' = 'FOO';
```

0

String Operations

CONCAT	Concatenation
UPPER	Converts string to uppercase.
LOWER	Converts string to uppercase.
TRIM	Removes spaces at the end of a string
LENGTH	Returns the length of the string.

***This is just a small sample of the string functions. See the DB docs for many more.

- MySQL: http://dev.mysql.com/doc/refman/5.7/en/string-functions.html
- PostgreSQL: https://www.postgresql.org/docs/9.5/static/functions-string.html

String Operations

```sql
SELECT LENGTH('foo');
```

?column?
3

```sql
SELECT UPPER('foo');
```

?column?
FOO

```sql
SELECT LOWER('FOO');
```

?column?
foo

```sql
SELECT  CONCAT('foo','bar');
```

?column?
foobar

```sql
SELECT CONCAT(TRIM('foo  '), 'bar');
```

?column?
foobar

LIKE: Simple String Pattern Matching

- **s LIKE p**: pattern matching on strings
- **p** may contain two special symbols:
  - **%** = any sequence of characters
  - **_** = any single character

LIKE: Simple String Pattern Matching

```
SELECT  'foo' LIKE 'foo';
```

?column?
t

```
SELECT  'foo' LIKE 'fo';
```

?column?
f

```
SELECT  'foo' LIKE '___';
```

?column?
t

3 underscores

```
SELECT  'foo' LIKE '_';
```

?column?
f

LIKE: Simple String Pattern Matching

```
SELECT  'foo' LIKE '%';
```

?column?
t

```
SELECT  'foo' LIKE CONCAT('%', 'f', '%');
```

?column?
t

```
SELECT  'foo' LIKE 'b%';
```

?column?
f

```
SELECT 'foo' LIKE CONCAT('%', x, '%')
FROM (VALUES ('f'))
AS t(x);
```

?column?
t

Query 39

Find all the last names that begin with “B”.

Query 39

Find all the last names that begin with “B”.

```
SELECT flast
FROM Faculty
WHERE flast LIKE 'B%';
```

Query 40

Find all the last names that contain an “e”.

Query 40

Find all the last names that contain an “e”.

```
SELECT flast
FROM Faculty
WHERE flast LIKE '%e%';
```

Query 41

Find all the last names that start with “d” and have a “B” in the 4th position.

Query 41

Find all the last names that start with “d” and have a “B” in the 4th position.

```
SELECT flast
FROM Faculty
   WHERE flast LIKE 'd__B%';
```

Query 42

List the names of employees who earn between $20,000 and $33,700 (inclusive).

Query 42

List the names of employees who earn between $20,000 and $33,700 (inclusive)

```
SELECT flast, ffirst, fmi
FROM Faculty
WHERE fsalary BETWEEN 20000 AND 33700;
```

or

```
SELECT flast, ffirst, fmi
FROM Faculty
WHERE fsalary >= 20000 AND fsalary <= 33700;
```

Temporal Math

- *datetime - datetime = interval*
- *datetime + interval = datetime*
- *datetime - interval = datetime*
- *interval + interval = interval*
- *interval - interval = interval*
- *interval + numeric = interval*
- *interval * numeric = interval*
- *interval / numeric = interval*

Temporal Math

- *datetime - datetime = interval*

```
SELECT '20100110'::DATE - '20090110'::DATE;
```

?column?
365

Temporal Math

- *datetime + interval = datetime*

```
SELECT '20100110'::DATE + 365;
```

?column?
2011-01-10

Temporal Math

- *datetime - interval = datetime*

```
SELECT '20100110'::DATE - 365;
```

?column?
2009-01-10

Pointless Complexity 1

```
SELECT DISTINCT fid
  FROM Department, Faculty;
```

reduces to

Pointless Complexity 1

```
SELECT DISTINCT fid
  FROM Department, Faculty;
```

reduces to

```
SELECT fid FROM Faculty;
```

Pointless Complexity 2

```
SELECT *
FROM Faculty
WHERE fid IN
  (SELECT fid FROM Faculty);
```

reduces to

Pointless Complexity 2

```
SELECT *
FROM Faculty
WHERE fid IN
   (SELECT fid FROM Faculty);
```

reduces to

```
SELECT * FROM Faculty;
```

Pointless Complexity 3

```
SELECT fid
FROM Faculty
GROUP BY fid
   HAVING COUNT(*) > 0;
```

reduces to

Pointless Complexity 3

```
SELECT fid
FROM Faculty
GROUP BY fid
   HAVING COUNT(*) > 0;
```

reduces to

```
SELECT fid
FROM Faculty;
```

Pointless Complexity 4

```
SELECT ffirst
FROM Faculty
GROUP BY ffirst;
```

reduces to

Pointless Complexity 4

```
SELECT ffirst
FROM Faculty
GROUP BY ffirst;
```

reduces to

```
SELECT DISTINCT ffirst
FROM Faculty;
```

SQL Seduction Summary

- DISTINCT compresses out duplicate rows
- Join tables based on their relationships
- Don’t accidentally undo a correlation
- NOT IN can not be rewritten as <>
- One IN with 2 columns not the same as two INs, one on each column
- Aggregate functions only look at non-null values
- Be mindful of GROUP BY syntax rules

As a reminder...

FACULTY

fid	flast	ffirst	fmi	fdept	fsalary	fmgr_id
12058	Borys	Ted	J	CSI	48000	22321
12206	Ryan	Alfred	C	ENG	48000	52110
21004	Perry	Bill	S	BIO	21800	31890
22321	Brady	Kathy	M	CSI	63400	52110
31890	Coulsen	Mary	*null*	BIO	21400	52110
32000	delBene	Bill	S	CSI	63500	22321
47862	Anders	John	P	ENG	33700	12206
52110	Smith	Alice	*null*	ADM	82000	*null*

Query 43

Display all the faculty columns with **fmi** equal to “S”.

```
SELECT *
FROM Faculty
WHERE fmi = “S”;
```

fid	flast	ffirst	fmi	fdept	fsalary	fmgr_id
21004	Perry	Bill	S	BIO	21800	31890
32000	delBene	Bill	S	CSI	63500	22321

Query 44

Display all the faculty columns with **fmi** not equal to “S”

```
SELECT *
FROM Faculty
WHERE fmi <> “S”;
```

fid	flast	ffirst	fmi	fdept	fsalary	fmgr_id
12058	Borys	Ted	J	CSI	48000	22321
12206	Ryan	Alfred	C	ENG	48000	52110
22321	Brady	Kathy	M	CSI	63400	52110
47862	Anders	John	P	ENG	33700	12206

Query 45

Combine all the rows from Query 43 and 44

```
SELECT * FROM Faculty WHERE fmi = “S”
UNION ALL
SELECT * FROM Faculty WHERE fmi <> “S”
```

Only 6 rows returned, not 8.

Query 46

Display all the faculty columns with unknown **fmi**.

```
SELECT *
FROM Faculty
WHERE fmi IS NULL;
```

Query 47

Combine all the rows from Query 43, 44, and 46.

```
SELECT * FROM Faculty WHERE fmi = “S”
UNION ALL
SELECT * FROM Faculty WHERE fmi <> “S”
UNION ALL
SELECT * FROM Faculty WHERE fmi IS NULL;
```

Query 48

Run this query to see why queries 43 through 47 work the way they do.

```
SELECT fmi,
                 fmi = 'S' AS "=S",
                fmi <> 'S' AS "<> S",
                (fmi IS NULL) AS "IS NULL",
                CHAR_LENGTH(fmi) AS length
FROM Faculty;
```

Note that true is 1 and false is 0.

Syntax valid for both Postgres and MySQL.

Query 49

Find faculty where fmi appears in fmi column in faculty.

```
SELECT *
FROM Faculty
WHERE fmi IN
   (SELECT fmi FROM Faculty);
```

Only 6 rows appear in the result.

Query 50

Find faculty where fmi doesn’t appear in fmi column in faculty.

```
SELECT *
FROM Faculty
WHERE fmi NOT IN
   (SELECT fmi FROM Faculty);
```

No rows appear in the result.

Query 51

Find faculty where fmi appears in fmi column in faculty using EXISTS construct.

```
SELECT * FROM Faculty AS a
WHERE EXISTS
   (SELECT b.fmi
    FROM Faculty AS b
    WHERE a.fid = b.fid);
```

All 8 rows appear in the result – never get empty set.

Query 51

Find faculty where fmi appears in fmi column in faculty using EXISTS construct.

```
SELECT * FROM Faculty AS a
WHERE EXISTS
   (SELECT b.fmi
    FROM Faculty AS b
    WHERE a.fid = b.fid);
```

Primary keys are guaranteed to never be null. Therefore, this condition must evaluate to true for one row. The result of the inner query may be a row with a null for its only attribute, but it is a row nonetheless.

All 8 rows appear in the result – never get empty set.

Query 52

Find faculty where fmi doesn’t appear in fmi column in faculty using EXISTS construct.

```
SELECT *
FROM Faculty AS a
WHERE NOT EXISTS
   (SELECT b.fmi
    FROM Faculty AS b
    WHERE a.fid = b.fid);
```

No rows appear in the result.

Query 53

Count the number of courses offered by each department, and include zero counts.

Query 53

Count the number of courses offered by each department, and include zero counts.

```
SELECT ddept, dname, COUNT(*) AS count
FROM Department, Course
WHERE ddept = cdept
GROUP BY ddept, dname;
```

ddept	dname	count
ATM	Atmospheric Science	2
BIO	Biology	2
CSI	Computer Science	3
ENG	English	1

Does not include the zero counts!

Query 53

Count the number of courses offered by each department, and include zero counts.

```
SELECT ddept, dname, COUNT(*) AS count
FROM Department, Course
WHERE ddept = cdept
GROUP BY ddept, dname
UNION
SELECT ddept, dname, 0
FROM Department WHERE ddept NOT IN
   (SELECT cdept FROM Course);
```

ddept	dname	count
ADM	Administration	0
ATM	Atmospheric Science	2
BIO	Biology	2
CSI	Computer Science	3
ENG	English	1
SPN	Spanish	0

Query 54

Rewrite Query 53 without the UNION.

Query 54

Rewrite Query 53 without the UNION.

```
SELECT ddept,
       dname,
       (SELECT COUNT(*)
        FROM Course
        WHERE ddept = cdept) AS count
FROM Department;
```

Query 55

Without using LIMIT, find the 3 highest salaries in faculty.

Query 55

Without using LIMIT, find the 3 highest salaries in faculty.

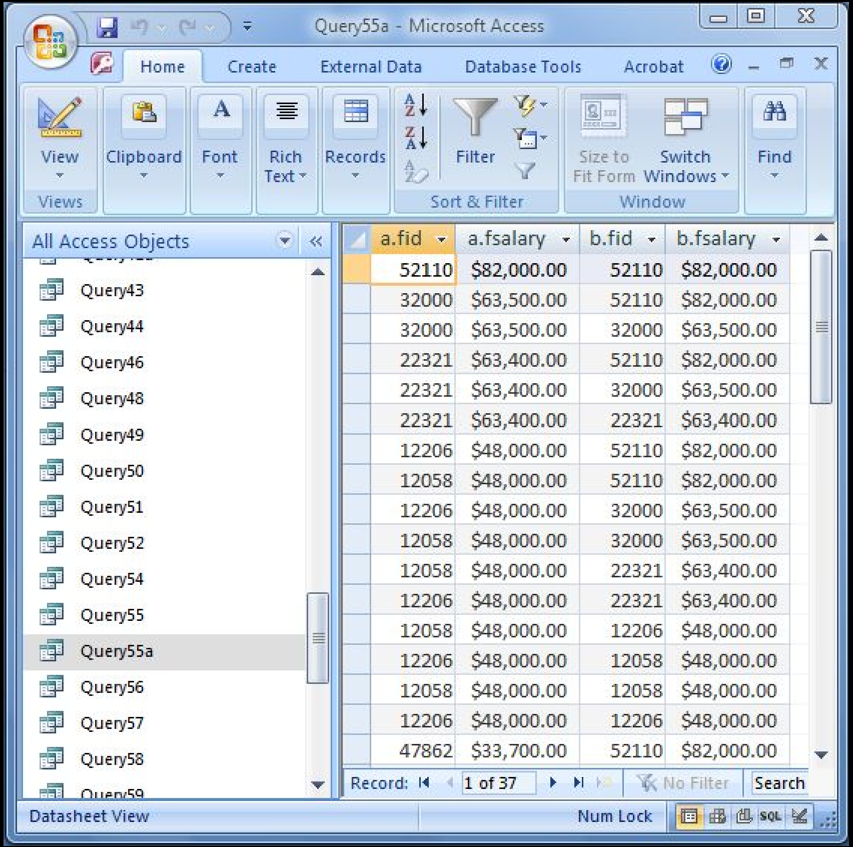
```
SELECT COUNT(*), a.fsalary
FROM Faculty AS a, Faculty AS b
WHERE a.fsalary <= b.fsalary
GROUP BY a.fsalary
HAVING COUNT(*) <= 3;
```

Query 55a

Consider:

```
SELECT a.fid, a.fsalary, b.fid, b.fsalary
FROM Faculty AS a, Faculty AS b
WHERE a.fsalary <= b.fsalary
ORDER BY a.fsalary DESC, b.fsalary DESC;
```

Query 55a Results



a.fid	a.fsalary	b.fid	b.fsalary
52110	$82,000.00	52110	$82,000.00
32000	$63,500.00	52110	$82,000.00
32000	$63,500.00	32000	$63,500.00
22321	$63,400.00	52110	$82,000.00
22321	$63,400.00	32000	$63,500.00
22321	$63,400.00	22321	$63,400.00
12206	$48,000.00	52110	$82,000.00
12058	$48,000.00	52110	$82,000.00
12206	$48,000.00	32000	$63,500.00
12058	$48,000.00	32000	$63,500.00
12058	$48,000.00	22321	$63,400.00
12206	$48,000.00	22321	$63,400.00
12058	$48,000.00	12206	$48,000.00
12206	$48,000.00	12058	$48,000.00
12058	$48,000.00	12058	$48,000.00
12206	$48,000.00	12206	$48,000.00
47862	$33,700.00	52110	$82,000.00

1

2

3

10

Query 56

Find the 6 highest salaries in faculty using the same strategy as Query 55.

Query 56

Find the 6 highest salaries in faculty using the same strategy as Query 55.

```
SELECT count(*), a.fsalary
FROM Faculty AS a, Faculty AS b
WHERE a.fsalary <= b.fsalary
GROUP BY a.fsalary
HAVING COUNT(*) <= 6;
```

count(*)	fsalary
6	33700
3	63400
2	63500
1	82000

Duplicate salary values highlight a problem.

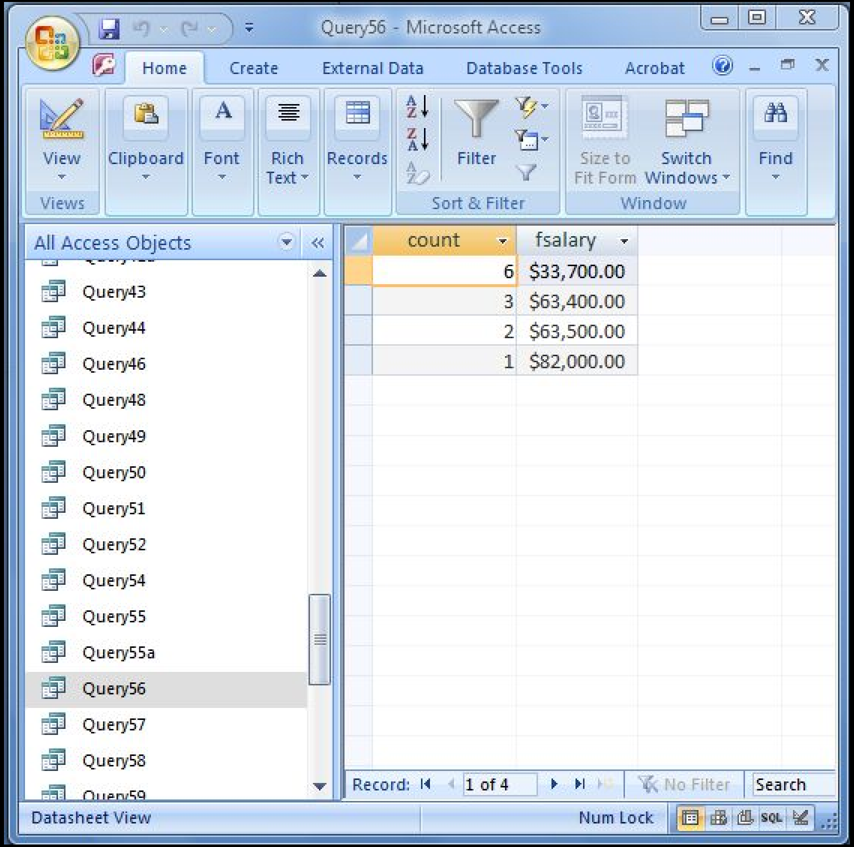
Query 56

Duplicate salary values highlight a problem.

```
SELECT fsalary, COUNT(*) as ct
FROM Faculty
GROUP BY fsalary
ORDER BY ct DESC, fsalary DESC;
```

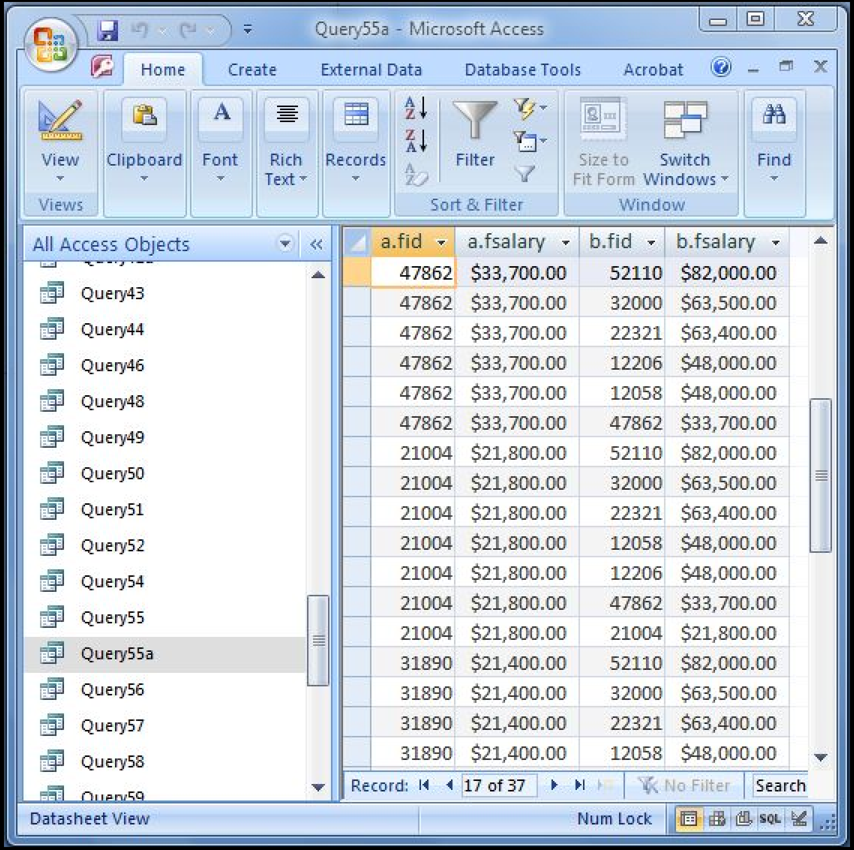
fsalary	count(*)
48000	2
82000	1
63500	1
63400	1
33700	1
21800	1
21400	1

Query 56 Results



count	fsalary
6	$33,700.00
3	$63,400.00
2	$63,500.00
1	$82,000.00

More of Query 55a Results



a.fid	a.fsalary	b.fid	b.fsalary
47862	$33,700.00	52110	$82,000.00
47862	$33,700.00	32000	$63,500.00
47862	$33,700.00	22321	$63,400.00
47862	$33,700.00	12206	$48,000.00
47862	$33,700.00	12058	$48,000.00
47862	$33,700.00	47862	$33,700.00
21004	$21,800.00	52110	$82,000.00
21004	$21,800.00	32000	$63,500.00
21004	$21,800.00	22321	$63,400.00
21004	$21,800.00	12058	$48,000.00
21004	$21,800.00	12206	$48,000.00
21004	$21,800.00	47862	$33,700.00
21004	$21,800.00	21004	$21,800.00
31890	$21,400.00	52110	$82,000.00
31890	$21,400.00	32000	$63,500.00
31890	$21,400.00	22321	$63,400.00
31890	$21,400.00	12058	$48,000.00

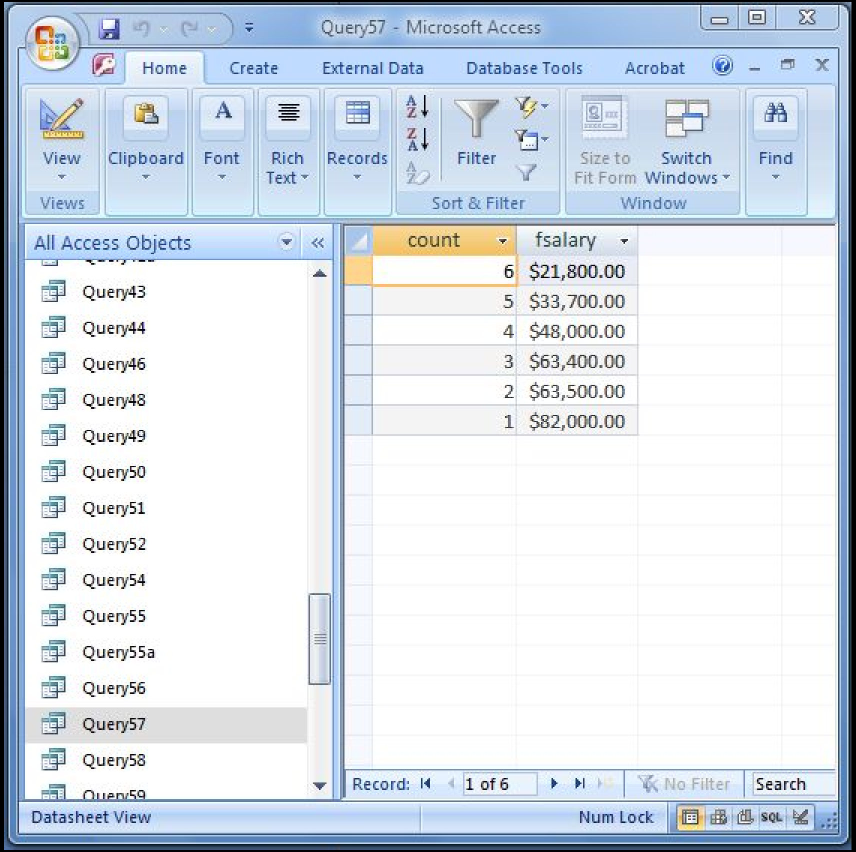
Record: 17 of 37

Query 57

Better way to find the 6 highest salaries in faculty.

```
SELECT COUNT(*), a.fsalary
FROM
  (SELECT DISTINCT a.fsalary, b.fsalary
   FROM Faculty AS a, Faculty AS b
   WHERE a.fsalary <= b.fsalary)
GROUP BY a.fsalary
HAVING COUNT(*) <= 6;
```

Query 57 Results



count	fsalary
6	$21,800.00
5	$33,700.00
4	$48,000.00
3	$63,400.00
2	$63,500.00
1	$82,000.00

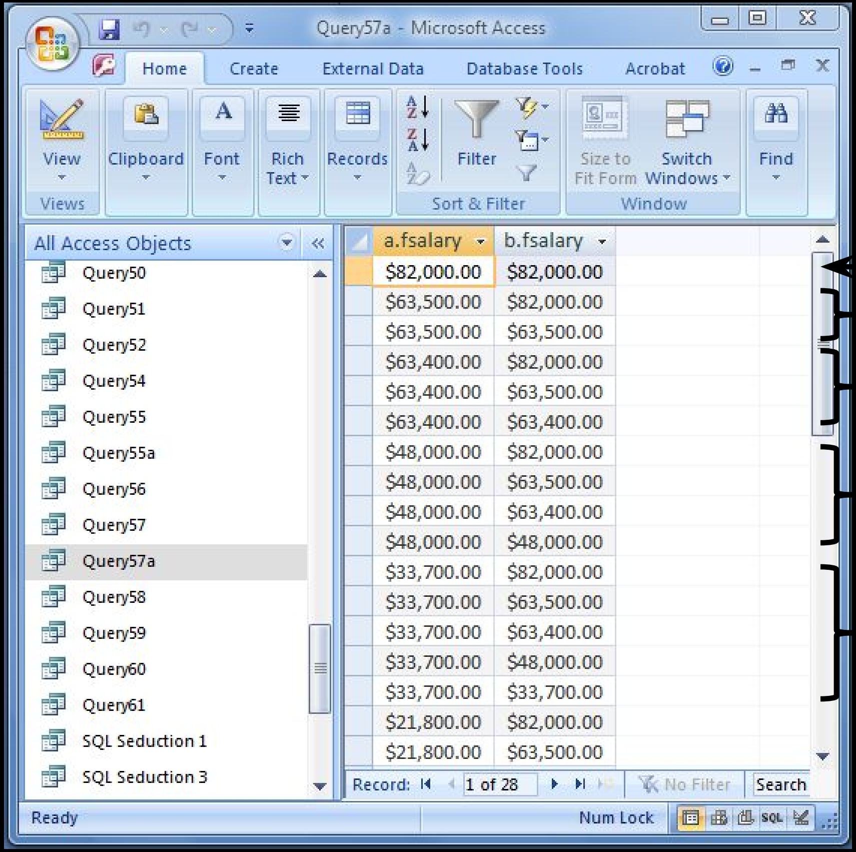
Record: 1 of 6

Query 57a

Consider:

```
SELECT DISTINCT a.fsalary, b.fsalary
FROM Faculty AS a, Faculty AS b
WHERE a.fsalary <= b.fsalary
ORDER BY a.fsalary DESC, b.fsalary DESC;
```

Query 57a Results



a.fsalary	b.fsalary
$82,000.00	$82,000.00
$63,500.00	$82,000.00
$63,500.00	$63,500.00
$63,400.00	$82,000.00
$63,400.00	$63,500.00
$63,400.00	$63,400.00
$48,000.00	$82,000.00
$48,000.00	$63,500.00
$48,000.00	$63,400.00
$48,000.00	$48,000.00
$33,700.00	$82,000.00
$33,700.00	$63,500.00
$33,700.00	$63,400.00
$33,700.00	$48,000.00
$33,700.00	$33,700.00
$21,800.00	$82,000.00
$21,800.00	$63,500.00

Join Types

- INNER JOIN
  - Data must match in both tables
- LEFT JOIN or LEFT OUTER JOIN
  - Data must match in both tables or appear in left table
- RIGHT JOIN or RIGHT OUTER JOIN
  - Data must match in both tables or appear in right table

Query 58

Original way to do an inner join

```
SELECT *
FROM Department, Faculty
WHERE ddept = fdept;
```

Query 59

Syntax introduced in SQL92 standard.

```
SELECT *
FROM Department INNER JOIN Faculty
ON ddept = fdept;
```

Query 60

Three-way table join:

```
SELECT *
FROM Department
  INNER JOIN Faculty
    ON ddept = fdept
  INNER JOIN Section
    ON fid = sid;
```

Query 61

Join department and faculty, and include departments with no faculty.

Query 61

Join department and faculty, and include departments with no faculty.

```
SELECT *
FROM Department
   LEFT JOIN Faculty
      ON ddept = fdept;
```

Query 62

Rewrite Query 61 as a RIGHT JOIN

```
SELECT *
FROM Faculty
    RIGHT JOIN Department
        ON ddept = fdept;
```

Full Outer Join

- Combine LEFT and RIGHT joins.

Full Outer Join

- MySQL

```
SELECT *
FROM t1 LEFT JOIN t2 ON t1.a = t2.a
UNION ALL
SELECT *
FROM t1 RIGHT JOIN t2 ON t1.a = t2.a;
```

a	a
1	1
2	null
null	null
1	1
null	null
null	3

t1
a
1
2
null

t2
a
1
null
3

Full Outer Join

- MySQL

```
SELECT *
FROM t1 LEFT JOIN t2 ON t1.a = t2.a
UNION
SELECT *
FROM t1 RIGHT JOIN t2 ON t1.a = t2.a;
```

a	a
1	1
2	null
null	null
null	3

t1
a
1
2
null

t2
a
1
null
3

Full Outer Join

- PostgreSQL

```
SELECT *
FROM t1 FULL OUTER JOIN t2
      ON t1.a = t2.a ;
```

a	a
1	1
2	null
null	null
null	3
null	null

t1
a
1
2
null

t2
a
1
null
3

Cautions

- Watch out for NULL values
- Subqueries can be used in many places
  - IN
  - EXISTS
  - FROM
  - WHERE
  - SELECT
- JOIN syntax introduced in SQL92 standard
- IN subquery syntax added in SQL99