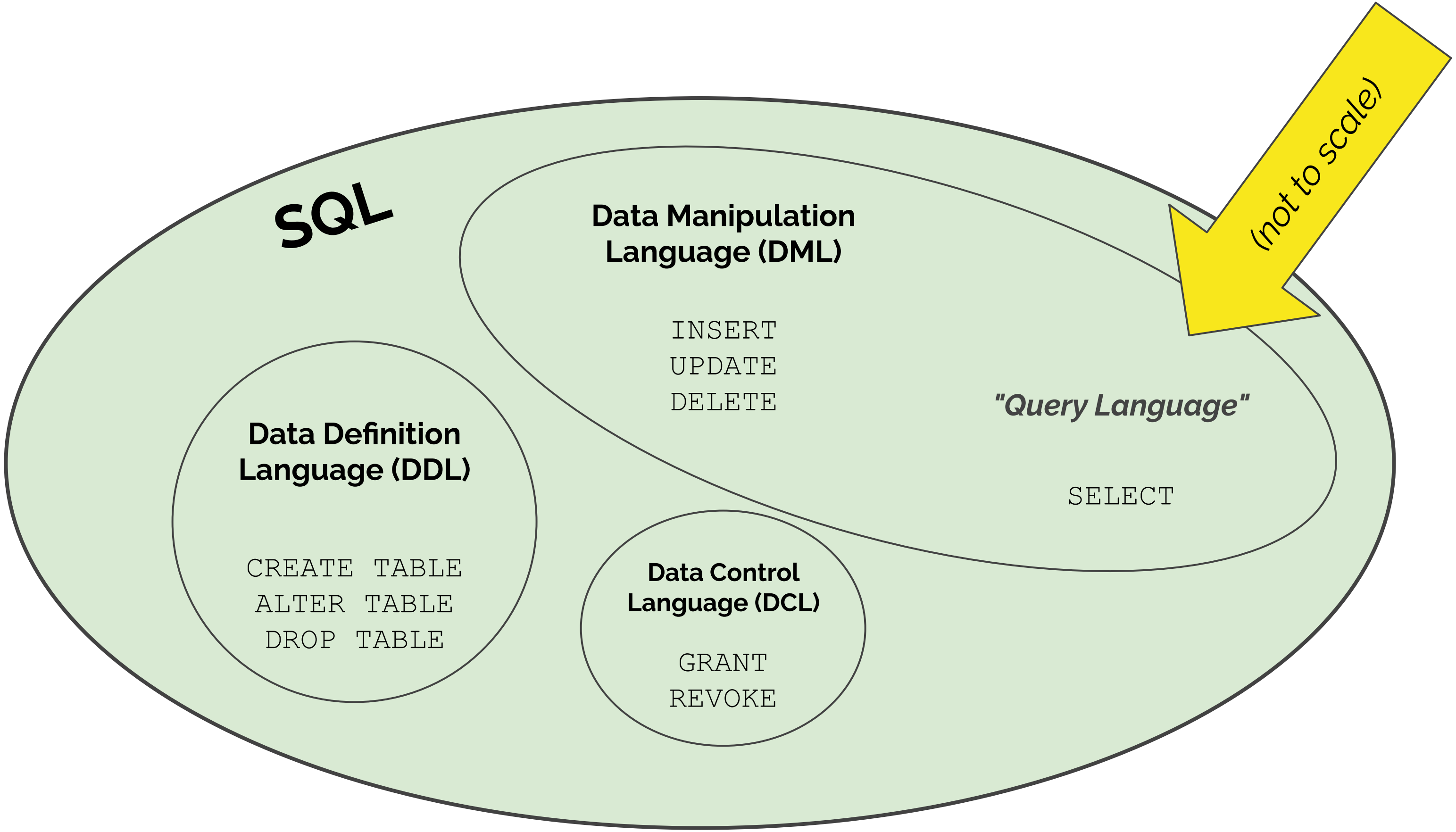
CSC 365

Introduction to Database Systems

Structured Query Language (SQL)



Relational Algebra

- Five core relational algebra operators can be combined to express interesting and complex queries
    - Selection
    - Projection
    - Cartesian Product
    - Union
    - Difference

- A database query language (ie. SQL) should be *at least* as expressive as relational algebra
    - SQL-92: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

SQL SELECT

Minimal syntax:

```
SELECT <list of columns>
FROM <table(s)>
[ WHERE <row filter expression> ]
;
```

Asterisk (*) can be used in place of column list to indicate all columns.

WHERE clause is optional

Sample Relation Instances

AIRPLANE

TailNum	**Make**	**Model**	**MaxSpeed**
C97W	Boeing	797	*null*
R53Q	Cessna	FG	220
T80H	Airbus	A380	634
G59K	Airbus	A320	450
P88T	Piper	Arrow	180
K30W	Boeing	707	450

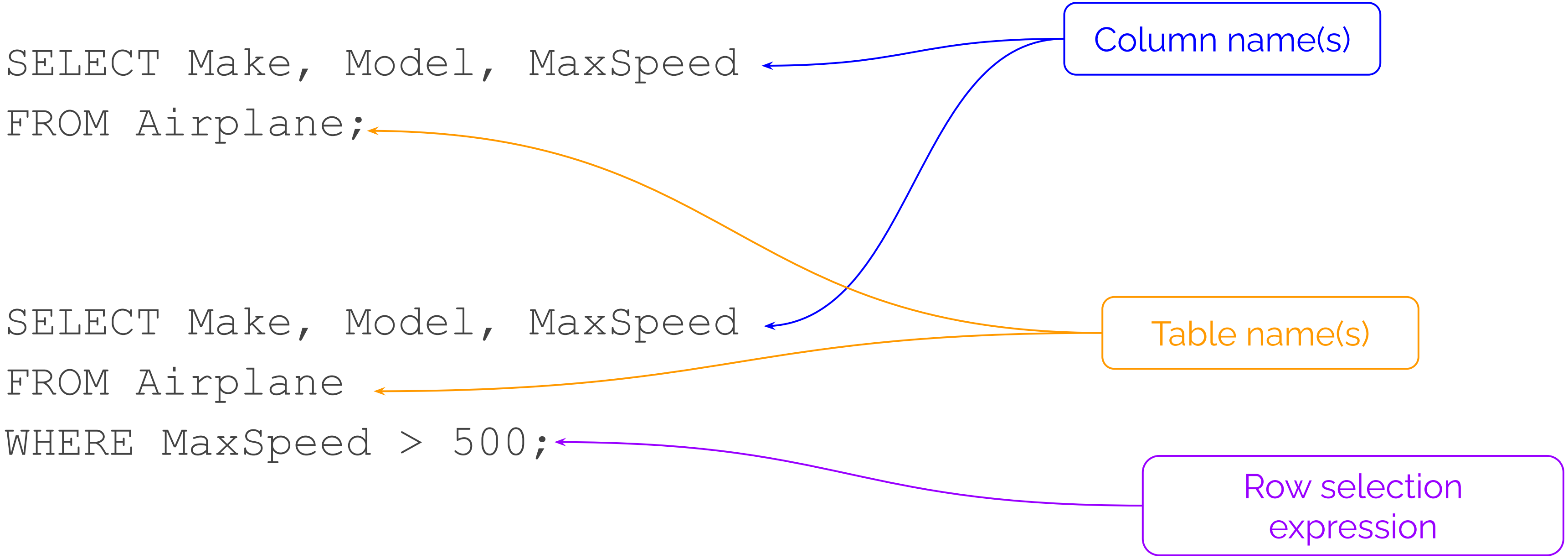
FLIGHT

TailNum	**PilotID**	**CopilotID**	**Runway**	**Date**
R53Q	K407	D342	S-2	9/1/17
T80H	K407	*null*	W-2	9/21/17
C97W	D342	*null*	W-2	8/9/21
T80H	D342	K407	W-3	9/9/17

PILOT

PilotID	**Name**
D342	Charlie
K407	Juliett
H452	Piper

SQL SELECT - Examples



```
SELECT Make, Model, MaxSpeed
FROM Airplane;
```

```
SELECT Make, Model, MaxSpeed
FROM Airplane
WHERE MaxSpeed > 500;
```

Column name(s)

Table name(s)

Row selection expression

SQL / Relational Algebra

Relational Algebra Operator		SQL
π	Projection	`SELECT DISTINCT`
×	Cartesian product	*comma* `or` `CROSS JOIN`
$\bowtie_\theta$	Theta join	`INNER JOIN ... ON`
σ	Selection	`WHERE`
ρ	Rename	`AS`

SQL SELECT vs Relational Algebra

Relational Algebra Expression	**SQL Statement**

$\sigma_{MaxSpeed > 500}(\text{AIRPLANE})$	SELECT *
	FROM Airplane
	WHERE MaxSpeed > 500

$\pi_{P.Name}(\rho_{P}(\text{PILOT}))$	SELECT DISTINCT P.Name
	FROM Pilot AS P

$\pi_{Date,Runway}(\sigma_{Date <= 9/1/2017\ AND\ Runway\ != \text{'W-3'}}(\text{FLIGHT}))$	SELECT DISTINCT Date, Runway
	FROM Flight
	WHERE Date <= 9/1/2017
	AND Runway <> 'W-3'

SQL SELECT vs Relational Algebra

Relational Algebra Expression	SQL Statement

RANK × SUIT	`` SELECT * FROM `Rank`, Suit; ``
	`` SELECT * FROM `Rank` CROSS JOIN Suit; ``

FLIGHT ⋈ PILOT	`SELECT *`
	`FROM Flight NATURAL JOIN Pilot`

$U \bowtie_{A < v.C} V$	`SELECT DISTINCT *`
	`FROM U, V`
	`WHERE A < V.C;`
	`-- or, using "infix" join syntax:`
	`SELECT DISTINCT *`
	`FROM U INNER JOIN V ON A < V.C;`

Refresher: Dates in SQL

The use of literal dates in SQL requires care.

Example (*as-is, this SQL statement will not produce the expected result*):

$\sigma_{\mathit{Date} <= 9/1/2017\ \mathit{AND}\ \mathit{Runway}\ != \ '\mathit{W}\text{-}3'}(\mathrm{FLIGHT})$

```
SELECT *
FROM Flight
WHERE Date <= 9/1/2017
  AND Runway != 'W-3'
```

Literal Dates in SQL

- Option 1: Use string literals in a recognized format
    - ANSI: `'YYYY-MM-DD HH:MM:SS'`
    - ISO 8601 date format allows you to specify time zone (ie. `'2017-10-05T17:47:17Z'`)
    - RDBMSs often have other default string representations of dates (typically, the default is configurable, so it's dangerous to rely on this!)

- Option 2: Use vendor-specific date functions
    - MySQL
        - MAKEDATE(<*year*>, <*day of year*>)
        - STR_TO_DATE(<string>, <format>) (format is C-style: %m, %d, %Y, etc.)

Date / Time Functions

A date value (literal or a table/column reference) may be used along with MySQL date functions to represent expressions such as: "within the last week", "last year", "how many days since ...?", etc.

```
SELECT DATE_ADD(DateEnrolled, INTERVAL 4 YEAR) AS TargetGradDate
FROM Student;

SELECT DateEnrolled, DATEDIFF(CURRENT_TIMESTAMP, DateEnrolled) / 365 AS YearsEnrolled
FROM Student
WHERE StudentID = '146461564';
```

Due to the use of vendor-specific functions, these two SQL statements are *MySQL-specific*, not ANSI SQL!

Literal Dates in SQL

Revised examples:

```
SELECT *
FROM Flight
WHERE Date = '2019-10-06';

SELECT *
FROM Flight
WHERE Date <= STR_TO_DATE('9/1/2019', '%m/%d/%Y') AND Runway != 'W-3';

SELECT *
FROM Flight
WHERE Date >= DATE_SUB(CURRENT_DATE, INTERVAL 4 MONTH);  -- within the last 4 months
```

SQL Scalar Functions

We will encounter a few more scalar functions in SQL, and will introduce them as needed.

There is little standardization in this area. For the most part, each RDBMS vendor supports its own unique collection of scalar functions.

Set Operators

	Relational Algebra	SQL
Set Union	$\cup$	`UNION`
Set Difference	$-$	`EXCEPT`
Set Intersection	$\cap$	`INTERSECT`

SQL SELECT - Union

- SQL defines two `UNION` variants:
    - `UNION` - Uses set rules (eliminates duplicates)
    - `UNION ALL` - Bag/multiset rules (preserves duplicate rows)

- Just as in relational algebra, SQL set operators (including `UNION` and `UNION ALL`) require *union compatibility:*

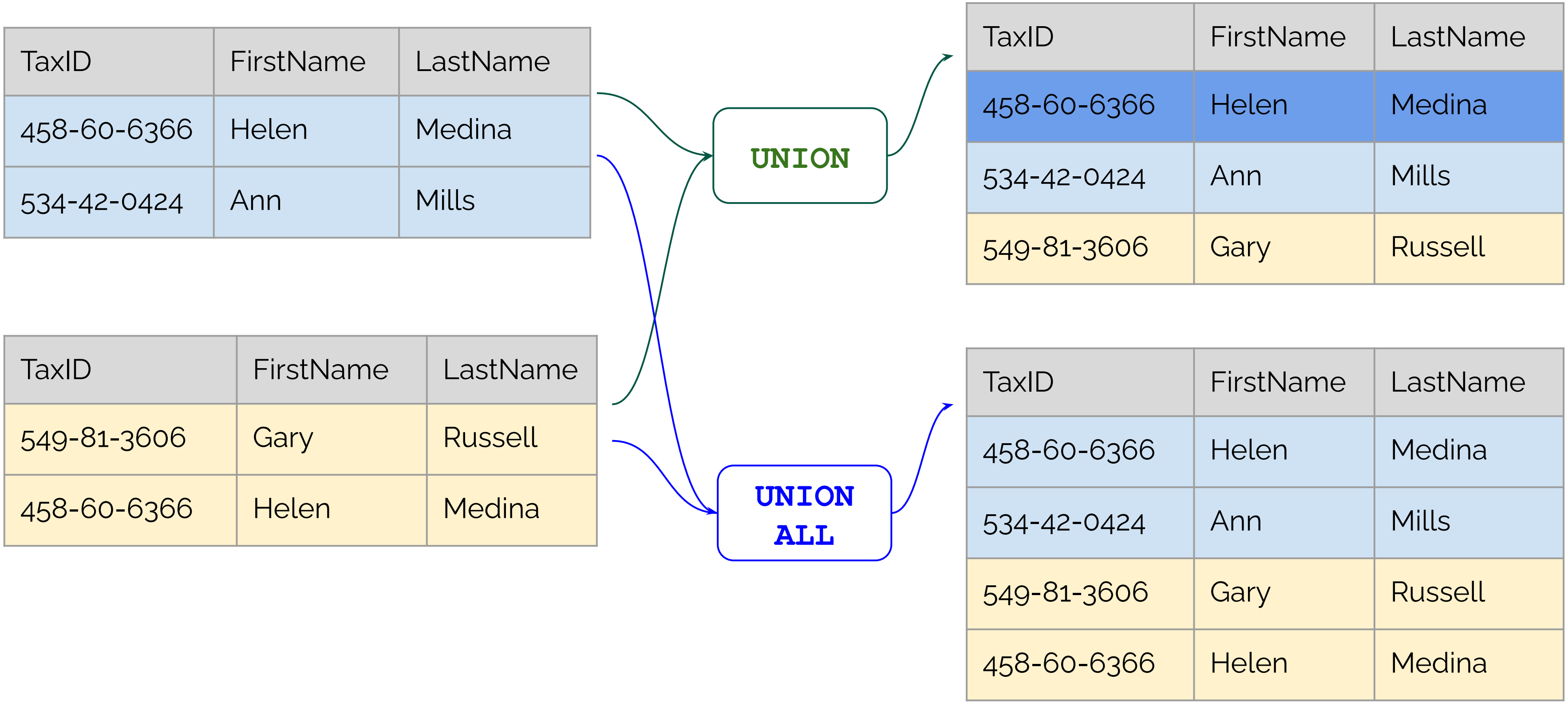
    Same number of columns and matching data types.

SQL SELECT - Union Examples

```
(SELECT Name FROM Pilot)
 UNION
(SELECT Make FROM Airplane)


(SELECT Name FROM Pilot)
 UNION ALL
(SELECT Make FROM Airplane)
```

UNION / UNION ALL



TaxID	FirstName	LastName
458-60-6366	Helen	Medina
534-42-0424	Ann	Mills

TaxID	FirstName	LastName
549-81-3606	Gary	Russell
458-60-6366	Helen	Medina

TaxID	FirstName	LastName
458-60-6366	Helen	Medina
534-42-0424	Ann	Mills
549-81-3606	Gary	Russell

TaxID	FirstName	LastName
458-60-6366	Helen	Medina
534-42-0424	Ann	Mills
549-81-3606	Gary	Russell
458-60-6366	Helen	Medina

SQL SELECT - INTERSECT

`INTERSECT` returns only rows that are present in *both* result sets.

```
-- List names of pilots that are also airplane makers
SELECT Name FROM Pilot
INTERSECT
SELECT Make as Name FROM Airplane
```

SQL SELECT - EXCEPT

`EXCEPT` returns rows that are present in the first query, but not present in the second query. Some vendors use the keyword `MINUS`

```
-- List IDs of pilots who do not have any flight records
SELECT PilotId FROM Pilot
EXCEPT
SELECT PilotId FROM Flight
```

SQL SELECT - Sets vs. Bags

- `UNION`, `EXCEPT`, and `INTERSECT` use *set semantics* by default, but the ANSI standard defines an `ALL` qualifier that preserves duplicates (bag or multiset semantics)
- Selection works the same on bags as it does on sets
- Projection with bag semantics *does not* eliminate duplicates
  - `SELECT` : bag semantics, no duplicate elimination!
  - `SELECT DISTINCT` : eliminates duplicate tuples
- Cartesian Product and joins operate on pairs of tuples. No difference in how they work when considering sets vs bags.

SQL SELECT - DISTINCT

Adding the `DISTINCT` keyword at the start of the `SELECT` clause causes only unique tuples to be returned (in other words: when you include `DISTINCT`, the result is a *set* of tuples rather than a multiset / bag)

```
SELECT DISTINCT Name
FROM Pilot
```

```
SELECT DISTINCT Make, Model
FROM Airplane
```

Relational Algebra - JOINs

- Join operators in relational algebra
    - Natural join
    - Theta join
    - Equijoin
    - Semijoin
    - Antijoin

- Selectively pair tuples from two relations.

- SQL supports the join operators listed above (*and a few others*)

SQL SELECT - Cartesian Product

There are two ways to express cartesian product in SQL.

Implicit (*often accidental!*):

```
SELECT *
FROM Pilot, Flight
```

Explicit:

```
SELECT *
FROM Pilot CROSS JOIN Flight
```

SQL SELECT - Natural Join

There are multiple ways to express a natural join in SQL.

Explicit projection and equijoin on common column(s):

```
SELECT DISTINCT Flight.*, Pilot.Name
FROM Flight, Pilot
WHERE Pilot.PilotID = Flight.PilotID
```

Implicitly, using `NATURAL JOIN`:

```
SELECT DISTINCT *
FROM Flight NATURAL JOIN Pilot
```

Why would we prefer one over the other?

Consider two relations:

EMP(EmpID, FirstName, LastName,
DeptID, Building, RoomNum)

DEPT(DeptID, Name, Building)

SQL SELECT - Natural Equijoin

The SQL standard defines `JOIN … USING` as a shortcut for natural equijoins. This allows explicit control over the join columns (vs. `NATURAL JOIN`)

```
SELECT *
FROM Emp INNER JOIN Dept USING (DeptID)
```

Consider the same two relations:

EMP(EmpID, FirstName, LastName,
DeptID, Building, RoomNum)

DEPT(DeptID, Name, Building)

SQL SELECT - Theta Join

We can use the following form to express theta joins:

Relational algebra expression:

$U \bowtie_{A < V.C \text{ AND } U.B \,!= V.B} V$

```
SELECT *
FROM U, V
WHERE A < V.C AND U.B <> V.B
```

ANSI SQL defines <> as the "not equal to" operator
Most RDBMSs support either <> or !=

SQL SELECT - Theta Join (Alternative)

As an alternative, we can use the following to express theta joins. Note the lack of a `WHERE` clause. Instead, we use the ANSI ***infix*** operator, `[INNER] JOIN … ON`, to specify our join condition. The `INNER` keyword is optional.

```
SELECT *
FROM U INNER JOIN V ON A < V.C AND U.B <> V.B
```

Relational algebra expression (same as previous slide):

$U \bowtie_{A < V.C \text{ AND } U.B \,!= V.B} V$

SQL SELECT - Theta Join - WHERE vs JOIN ... ON

Two ways to express theta join:

Same condition

```
SELECT * FROM U, V WHERE A < V.C AND U.B <> V.B


SELECT * FROM U INNER JOIN V ON A < V.C AND U.B <> V.B
```

- What's the difference?
  - `JOIN ON` can help prevent accidental cartesian products
  - Readability, especially with more than one join (separates relationship/join logic from selection logic)
  - Other types of joins we'll see in SQL (`OUTER`)

Recommendation: `JOIN` ... `ON` for conditions used to pair tuples,
`WHERE` clause for conditions that "filter" tuples

SQL SELECT - Self JOIN

A table may be of joined to itself (*table aliases required*)

Find hard-disk sizes that occur in two or more PCs.

$\boldsymbol{\pi}_{hd}(\sigma_{\text{PC1.model != PC2.model ^ PC1.hd = PC2.hd}}(\varrho_{\text{PC1}}(\text{PC}) \times \varrho_{\text{PC2}}(\text{PC})))$

```
SELECT DISTINCT pc1.hd
FROM pc AS pc1, pc AS pc2
WHERE pc1.model <> pc2.model AND pc1.hd = pc2.hd

-- JOIN … ON syntax remains an option
```

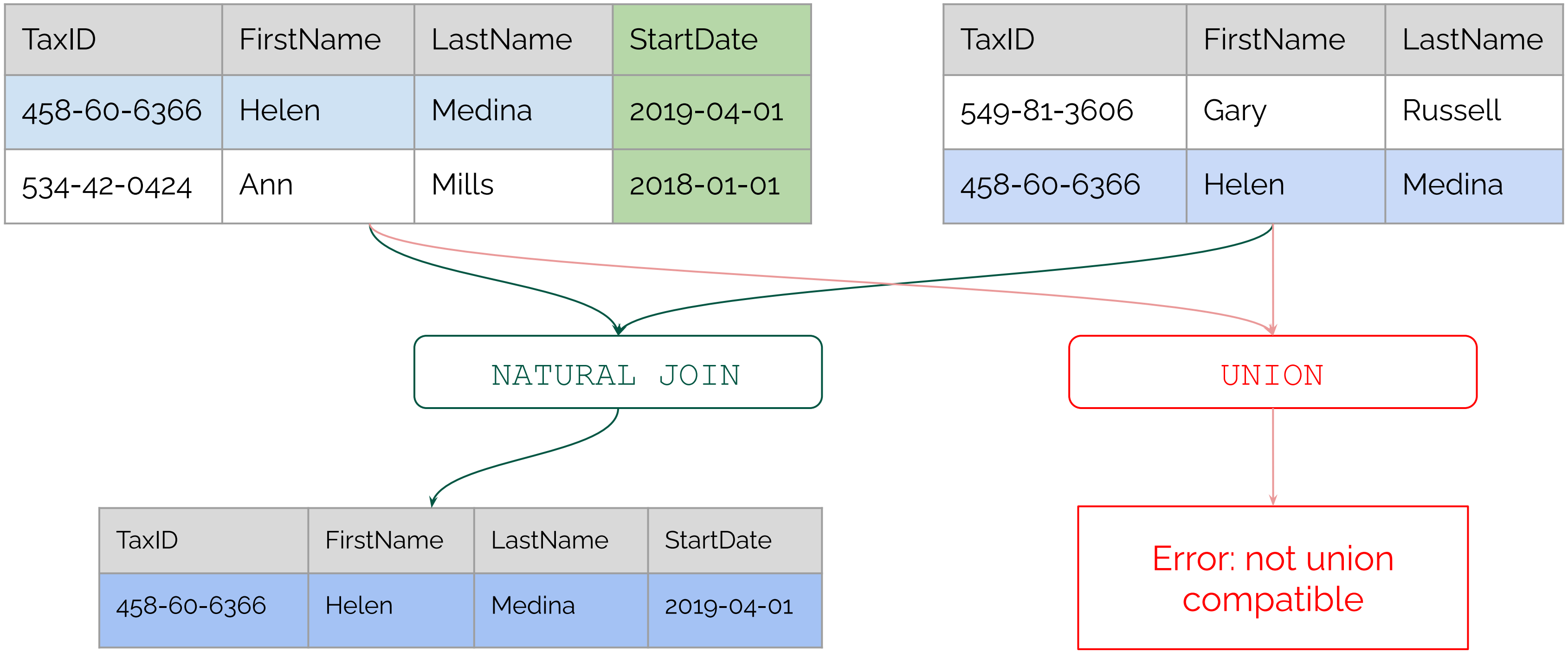
JOIN or UNION?

Joins and unions both combine data from multiple tables, but they do so in different ways.

Informally, joins combine *columns* while unions operate on entire *rows*.

Important to keep in mind behavior with empty tables.

Set Operations vs NATURAL JOIN



TaxID	FirstName	LastName	StartDate
458-60-6366	Helen	Medina	2019-04-01
534-42-0424	Ann	Mills	2018-01-01

TaxID	FirstName	LastName
549-81-3606	Gary	Russell
458-60-6366	Helen	Medina

NATURAL JOIN

TaxID	FirstName	LastName	StartDate
458-60-6366	Helen	Medina	2019-04-01

UNION

Error: not union compatible

SQL SELECT - Multiple JOINs

A single `SELECT` statement may include any number of `JOIN`s:

```
SELECT Pilot.Name, Airplane.Make, Airplane.Model, Airplane.TailNum, Flight.Date
FROM Pilot
   INNER JOIN Flight ON Pilot.PilotID = Flight.PilotID
   INNER JOIN Airplane ON Airplane.TailNum = Flight.TailNum
```

For *inner joins*, the join order has no impact on results.

SQL SELECT - JOIN & WHERE

A `SELECT` statement may combine `JOIN` and `WHERE` (and a few more clauses that we will discuss shortly):

```
SELECT DISTINCT P.Name AS Pilot, A.Make, A.Model, A.TailNum, F.Date
FROM Pilot P
  INNER JOIN Flight F ON P.PilotID = F.PilotID
  INNER JOIN Airplane A ON A.TailNum = F.TailNum
WHERE F.Date <= '2019-12-31' AND A.Make <> 'Boeing' AND MaxSpeed > 500
```

Corresponding Relational Algebra expression?

SQL - Precedence of Logical Operators

Logical operators in SQL adhere to the following precedence rules:

`AND` takes precedence over `OR`, `NOT` takes precedence over both

Parentheses are advised. **DeMorgan's laws** are handy to keep in mind when dealing with complex expressions:

```
NOT(A OR B) = NOT(A) AND NOT(B)          NOT(A AND B) = NOT(A) OR NOT(B)
```

SQL - String Pattern Matching

ANSI SQL supports a simple string matching operator:

```
<string> LIKE <pattern>
```

<pattern> may include wildcards: `%` (zero or more characters) or _ (one character)

```
SELECT *
FROM Pilot
WHERE Name LIKE 'Vic%'
```

SQL - String Pattern Matching

Also:

```
<string> NOT LIKE <pattern>
```

```
SELECT *
FROM Pilot
WHERE Name NOT LIKE '%_r'
```

Warning: In MySQL, `LIKE` and `NOT LIKE` are **case INsensitive** by default!

Some RDBMSs allow more advanced pattern matching (for example: MySQL `REGEX / NOT REGEX`) However, this is not part of the ANSI standard.

SQL NULL

NULL handling requires care in SQL. Specialized comparison operators:

```
IS NULL
IS NOT NULL
```

```
SELECT *
FROM Airplane
WHERE MaxSpeed IS NOT NULL
```

None of these will work as expected:

```
WHERE MaxSpeed = NULL

WHERE NOT(MaxSpeed = NULL)

WHERE MaxSpeed != NULL
```

SQL - NULLs

```
SELECT *
FROM Airplane
WHERE MaxSpeed > 250 OR MaxSpeed <= 250
```

```
SELECT *
FROM Airplane
WHERE (MaxSpeed = 220) OR NOT (MaxSpeed = 220)
```

SQL NULL - Truth Table

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

SQL - Sorting Results

To control the order in which tuples are returned, we may append the optional ORDER BY clause to a SELECT statement. ORDER BY may include one or more attributes.

```
SELECT *
FROM Airplane
ORDER BY Make, Model
```

By default, order is lowest to highest for each attribute. We can reverse the order (per attribute) by appending DESC to any attribute.

SQL - Expressions in SELECT

SQL supports a variety of expressions, including basic arithmetic and string/date scalar functions. Examples:

```
SELECT Make, Model, MaxSpeed * 1.60934 AS SpeedKPH
FROM Airplane
```

```
SELECT Name, CHAR_LENGTH(Name) AS NameLength
FROM Pilot
ORDER BY CHAR_LENGTH(Name)
```

`CHAR_LENGTH()` counts characters — taking into account multi-byte character sets — `LENGTH()` counts *bytes*

SQL - Expressions in SELECT

More examples:

`CONCAT()` and `DATE_SUB()` are *MySQL-specific* functions!

```
SELECT CONCAT(Make, ' ', Model, '/', TailNum) AS PlaneDescription
FROM Airplane
WHERE CONCAT(Make, ' ', Model) LIKE '%Air%' OR MaxSpeed IS NULL
```

```
SELECT Date, DATE_SUB(Date, INTERVAL 1 DAY) AS FlightPlanDue
FROM Flight
WHERE Date <= CURRENT_DATE
ORDER BY Date DESC
```

WHERE Clause

We have seen basic comparison and boolean connectors in the `WHERE` clause. SQL also supports the following optional syntax for convenience:

SQL Condition	Equivalent to...
`column BETWEEN x AND y`	`column >= x AND column <= y`
`column NOT BETWEEN x AND y`	`column < x OR column > y`
`column IN (a, b, c)`	`column = a OR column = b OR column = c`
`column NOT IN (a, b, c)`	`column <> a AND column <> b AND column <> c`

SELECT Statement - Logical Processing Order

3	SELECT
4	DISTINCT
1	FROM … JOIN … ON
2	WHERE
5	UNION / INTERSECT / EXCEPT
6	ORDER BY

SQL / Relational Algebra

SQL	Relational Algebra Operator	
`SELECT DISTINCT`	π	Projection
`FROM`	×	Cartesian product
`INNER JOIN ... ON`	$\bowtie_\theta$	Theta join
`WHERE`	σ	Selection
`AS`	ρ	Rename
`UNION`	$\cup$	Set Union
`EXCEPT`	$-$	Set Difference
`INTERSECT`	$\cap$	Set Intersection

SQL SELECT vs Relational Algebra

- The SQL language fully supports the five core relational algebra operators and derived operators (joins, intersect). We will introduce a few additional tools, including:
    - Additional join variations
    - Grouping / aggregation (`GROUP BY`)

WHERE Clause - Anywhere

The WHERE clause can be used with SELECT (as we have seen). The same WHERE clause syntax applies to UPDATE, and DELETE.

```
SELECT *
FROM Flight
WHERE Date > CURRENT_DATE AND Runway LIKE 'W%';
```

```
DELETE
FROM Flight
WHERE Date > CURRENT_DATE AND Runway LIKE 'W%';
```

UPDATE

The UPDATE statement can be used to change multiple columns in multiple rows of a *single* table.

Syntax:

```
UPDATE <table>
SET <column1> = <value1> [, <column2> = <value2>]
[WHERE <predicate>];
```

UPDATE Example

All flights that depart on a future date from a runway that begins with "W" now depart on the following day.

```
UPDATE Flight
SET Date = DATE_ADD(Date, INTERVAL 1 DAY)
WHERE Date > CURRENT_DATE AND Runway LIKE 'W%';
```

UPDATE Example #2

A recent software update has increased the maximum speed of all Boeing aircraft by 10%. Also, all Boeing model numbers are now suffixed with an "s"

```
UPDATE Airplane
SET MaxSpeed = MaxSpeed * 1.1, Model = CONCAT(Model, 's')
WHERE Make  = 'Boeing';
```

UPDATE Example #3

All airplanes must undergo regular inspections. Add a new column to the Airplane table to record the last date of inspection. Set this value to January 29, 2024 for all airplanes.

```
ALTER TABLE Airplane ADD COLUMN LastInspected DATE;
```

```
UPDATE Airplane
SET LastInspected = '2023-04-26';
```

No `WHERE` clause, ***all*** rows will be updated!

Further Schema Changes

In addition to the inspection date, we need to store the initials of the mechanic who performed each inspection, as well as any notes he/she may have recorded.

To add a foreign key constraint:

```
ALTER TABLE <table>
   ADD [CONSTRAINT [name of constraint]]
   FOREIGN KEY (<column>, ...)
   REFERENCES <table> (<column>,...);
```

To drop a column:

```
ALTER TABLE <table> DROP [COLUMN] <column>;
```

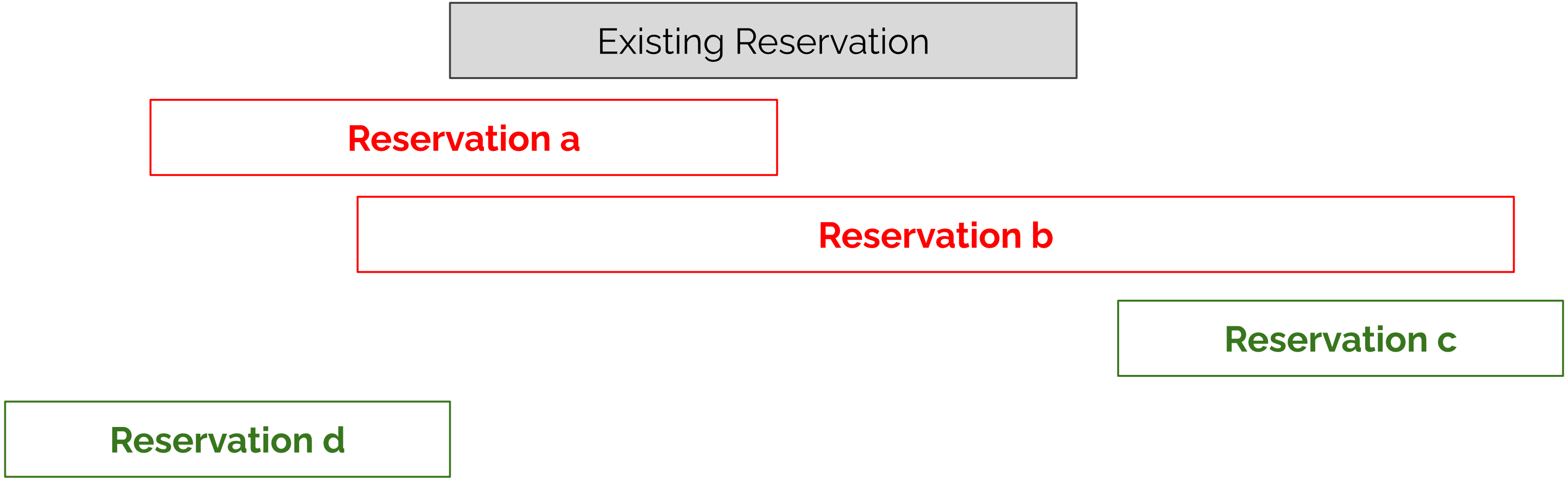
Example: Overlapping Dates

Given a relation with schema:

RESERVATIONS(Code, Room, CheckIn, CheckOut)

How could we identify reservations that would overlap for a given date range? (ie. where at least one night would be double-booked for the room)

Example: Overlapping Dates



To avoid overlap, a reservation must (for every existing reservation) start after the "other" reservation ends or end before the "other" reservation begins.