MLDS-413 Introduction to Databases and Information Retrieval

Lecture 12
Set Operations, `CASE` statements, and Regular Expressions

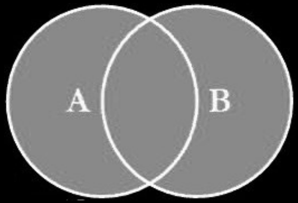
Instructor: Nikos Hardavellas

Slides adapted from Steve Tarzia

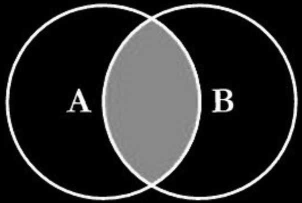
Last Lecture

- `NATURAL JOIN`s have an implicit `ON` clause matching columns with the same name
  - This is a good motivation to use consistent column names
  - Can be used for both `INNER` and `LEFT JOIN`s
- `LEFT JOIN`s keep unmatched rows from the left table
  - In the result, unmatched rows will have *`NULL`*s on the right-hand side
  - Useful when supplementing optional data from another table
- `EXCEPT` excludes rows matching a `SELECT` statement

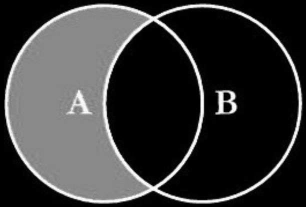
UNION, INTERSECT, and EXCEPT are used to combine two SELECT statements



- **UNION** prints rows from *either of two* SELECTs (printing duplicates just once)



- **INTERSECT** prints rows *present in both* SELECTs

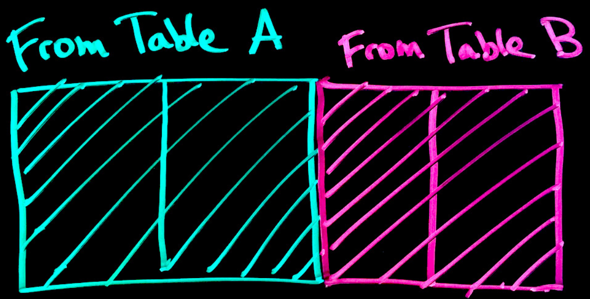


- **EXCEPT** prints rows *present in one* SELECT but *missing from another* SELECT

JOIN vs. UNION

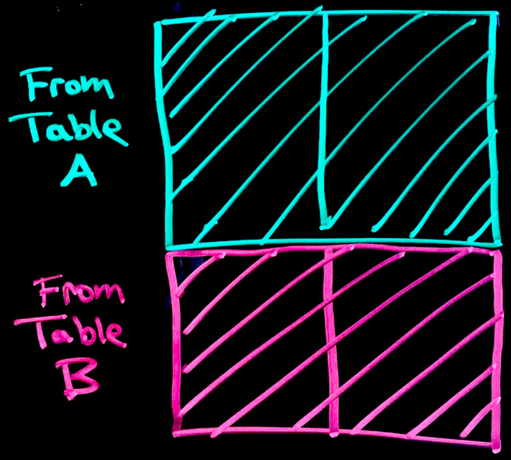
- JOINs combine tables *horizontally*
  - Creates a wider set of rows, with columns from both tables
  - Rows from two tables **may** be matching on one or more columns
    - But, they **do not have** to match (e.g., JOIN without ON clause)

JOIN:



- UNION, INTERSECT, and EXCEPT combine result tables *vertically*
  - Changes the number of *rows*, not columns
  - Number & type of columns in the two result tables must match

UNION:



Combining SELECTs through UNION, INTERSECT, and EXCEPT

- Operate on *union-compatible* queries: the left and right SELECT queries must
  1. Return the same number of columns
  2. The matching columns must have compatible data types
- UNION prints all rows from both left and right selects
  - Example: "List the names of all Customers and Employees"

```
SELECT CustFirstName FROM Customers
UNION
SELECT EmpFirstName FROM Employees;
```

  - Duplicates are printed just once
- INTERSECT prints only rows from the left and right SELECTs that match
  - Example: "Which first names are common among students and staff"?

```
SELECT StfFirstName FROM Staff
INTERSECT
SELECT StudFirstName from Students;
```

Misuses of UNION, INTERSECT, and EXCEPT

- Each SELECT statement gets data from a *different set of tables*
  - Otherwise it would be easier to just use a WHERE clause

```
SELECT * FROM Staff WHERE name="Jane”
  UNION SELECT * FROM Staff WHERE name="John”
```

simplify to:

```
SELECT * FROM Staff WHERE name="Jane" OR name="John”
```

```
SELECT * FROM Student_Schedules NATURAL JOIN Students
EXCEPT
SELECT * FROM Student_Schedules NATURAL JOIN Students
  WHERE Grade IS NULL
```

simplify to:

```
SELECT * FROM Student_Schedules NATURAL JOIN Students
  WHERE Grade IS NOT NULL
```

CASE conditional

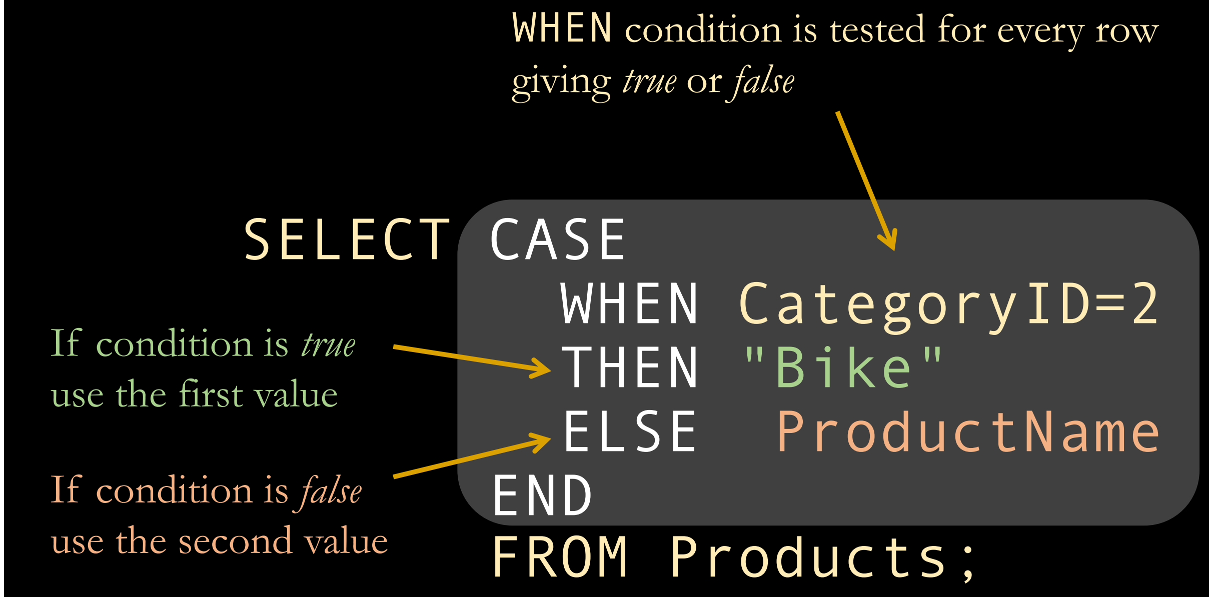
- Many programming languages have `if … then … else …` expressions
- Example in C language: `var = cond ? 10 : 20 ;`

- SQL's equivalent is CASE:

```
CASE WHEN … THEN … ELSE … END
```

- Condition after WHEN is checked for true/false (1/0)
  - If the condition is true, then the expression after THEN is used
  - Otherwise (if the condition is false), then the expression after ELSE is used

CASE in more detail



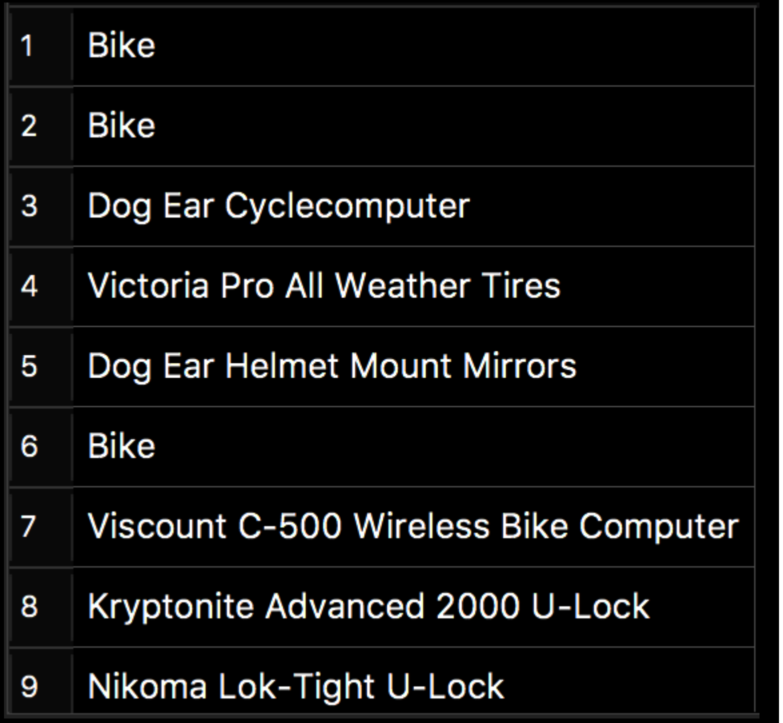
WHEN condition is tested for every row giving *true* or *false*

```
SELECT CASE
         WHEN CategoryID=2
         THEN "Bike"
         ELSE  ProductName
       END
       FROM Products;
```

If condition is *true* use the first value

If condition is *false* use the second value

Output:

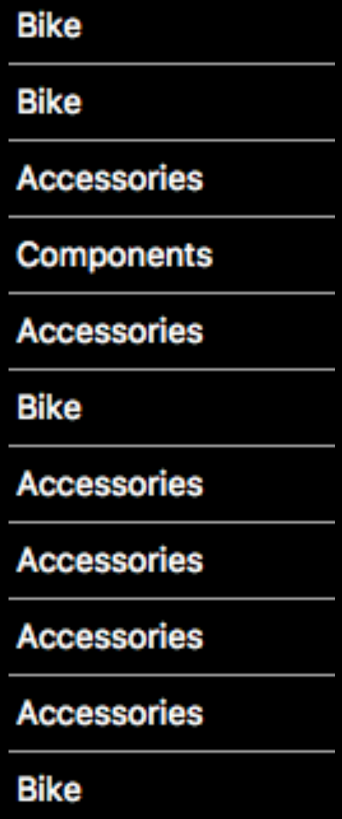


1	Bike
2	Bike
3	Dog Ear Cyclecomputer
4	Victoria Pro All Weather Tires
5	Dog Ear Helmet Mount Mirrors
6	Bike
7	Viscount C-500 Wireless Bike Computer
8	Kryptonite Advanced 2000 U-Lock
9	Nikoma Lok-Tight U-Lock

CASE with many “cases”

```
SELECT CASE
     WHEN CategoryID=1 THEN "Accessories"
     WHEN CategoryID=2 THEN "Bike"
     WHEN CategoryID=3 THEN "Clothing"
     WHEN CategoryID=4 THEN "Components"
     WHEN CategoryID=5 THEN "Racks"
     WHEN CategoryID=6 THEN "Tires"
     ELSE ProductName
END
FROM Products;
```

Output:



```
Bike
Bike
Accessories
Components
Accessories
Bike
Accessories
Accessories
Accessories
Accessories
Bike
```

Combining CASE statements

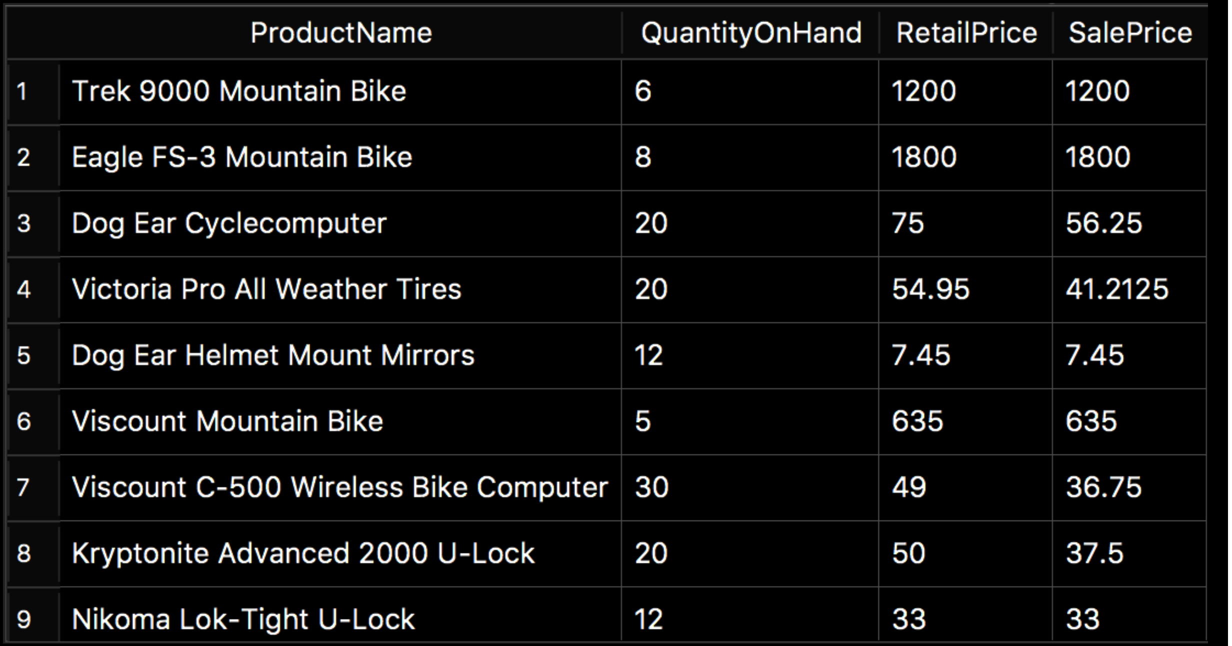
- “Print firstName for children or Mr./Mrs. lastName for adults”

```
SELECT
  CASE WHEN age<18
  THEN firstName
  ELSE (CASE WHEN gender="male"
             THEN "Mr. "
             ELSE "Mrs. "
             END
        || lastName)
  END
FROM people;
```

Another CASE example

Let's say we want to print "sale prices" for products that are overstocked. Any products with 20 or more items in stock are discounted 25%

```
SELECT ProductName,
       QuantityOnHand,
       RetailPrice,
CASE
  WHEN QuantityOnHand >= 20
  THEN 0.75*RetailPrice
  ELSE RetailPrice
END
   AS SalePrice
FROM Products
```



	ProductName	QuantityOnHand	RetailPrice	SalePrice
1	Trek 9000 Mountain Bike	6	1200	1200
2	Eagle FS-3 Mountain Bike	8	1800	1800
3	Dog Ear Cyclecomputer	20	75	56.25
4	Victoria Pro All Weather Tires	20	54.95	41.2125
5	Dog Ear Helmet Mount Mirrors	12	7.45	7.45
6	Viscount Mountain Bike	5	635	635
7	Viscount C-500 Wireless Bike Computer	30	49	36.75
8	Kryptonite Advanced 2000 U-Lock	20	50	37.5
9	Nikoma Lok-Tight U-Lock	12	33	33

CASE can also be used in filters

Print customers named “Martin” but refer to the first name in the friendly state of Illinois and the last name elsewhere

```
SELECT * FROM Customers
WHERE   CASE
          WHEN CustState = "IL"
          THEN CustFirstName
          ELSE CustLastName
        END
      = "Martin"
```

Incidentally, this is equivalent to:

```
SELECT * FROM Customers WHERE
  (CustState = "IL" AND CustFirstName = "Martin")
  OR (CustState != "IL" AND CustLastName = "Martin");
```

Tell me if each recipe is vegetarian, and if not, then name one of its meat/seafood ingredients

Print a different message for veg/meat recipes

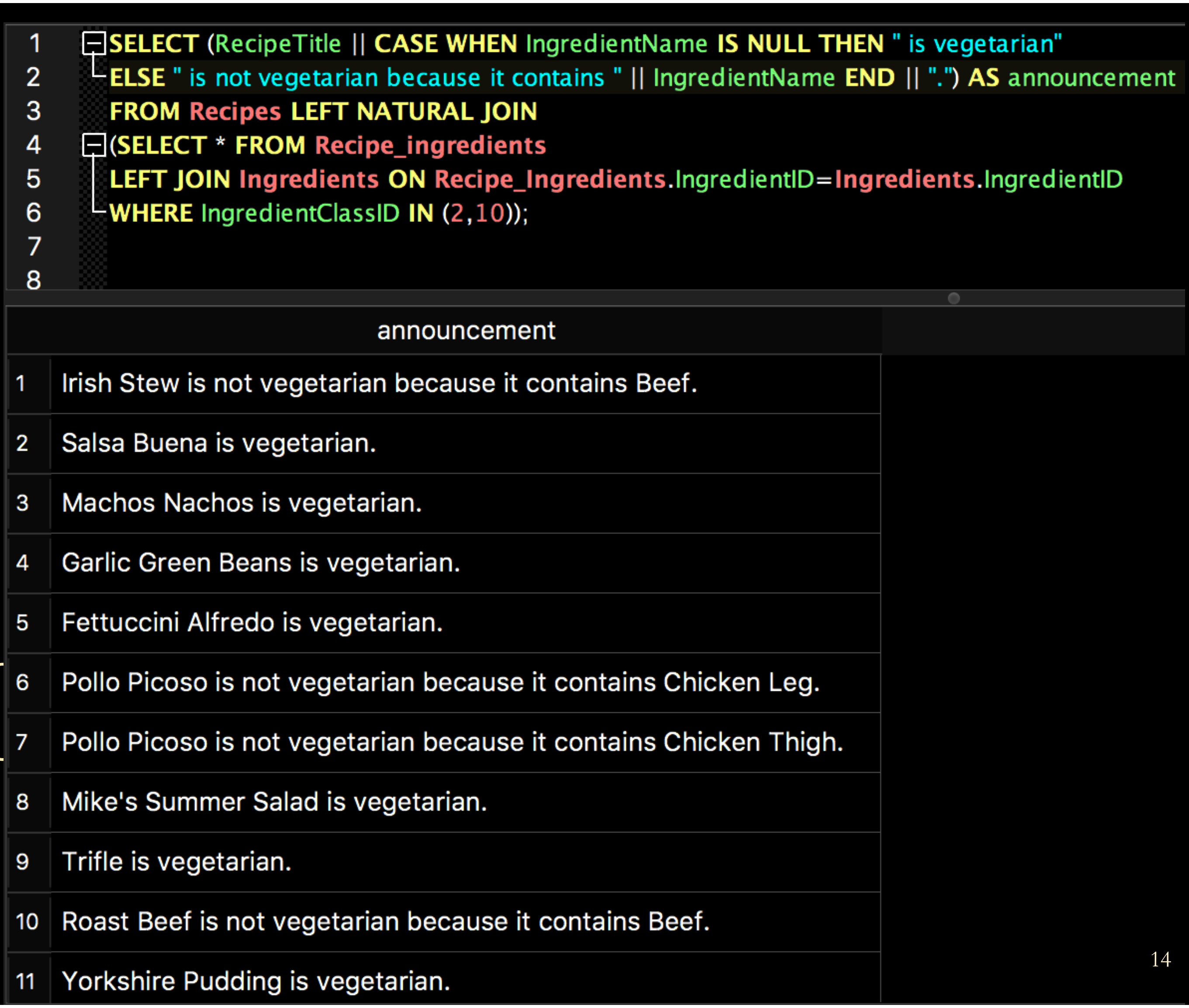
```
SELECT (RecipeTitle ||
  CASE WHEN IngredientName IS NULL THEN " is vegetarian"
  ELSE " is not vegetarian because it contains "
       || IngredientName END || ".") AS announcement
FROM Recipes LEFT NATURAL JOIN
(SELECT * FROM Recipe_Ingredients
  JOIN Ingredients ON
  Recipe_Ingredients.IngredientID=Ingredients.IngredientID
  WHERE IngredientClassID IN (2,10));
```

`LEFT JOIN` with a table printing only the meat/seafood recipe ingredients

Meat or seafood

* Note that a `NATURAL JOIN` cannot be used between Recipe_Ingredients and Ingredients because they have two columns in common (IngredientID and MeasureAmountID) and MeasureAmountID does not always match

The result:



```sql
SELECT (RecipeTitle || CASE WHEN IngredientName IS NULL THEN " is vegetarian"
ELSE " is not vegetarian because it contains " || IngredientName END || ".") AS announcement
FROM Recipes LEFT NATURAL JOIN
(SELECT * FROM Recipe_ingredients
LEFT JOIN Ingredients ON Recipe_Ingredients.IngredientID=Ingredients.IngredientID
WHERE IngredientClassID IN (2,10));
```

	announcement
1	Irish Stew is not vegetarian because it contains Beef.
2	Salsa Buena is vegetarian.
3	Machos Nachos is vegetarian.
4	Garlic Green Beans is vegetarian.
5	Fettuccini Alfredo is vegetarian.
6	Pollo Picoso is not vegetarian because it contains Chicken Leg.
7	Pollo Picoso is not vegetarian because it contains Chicken Thigh.
8	Mike's Summer Salad is vegetarian.
9	Trifle is vegetarian.
10	Roast Beef is not vegetarian because it contains Beef.
11	Yorkshire Pudding is vegetarian.

Could improve the query to eliminate this duplication

Query without duplication – the sneaky version

```
SELECT (RecipeTitle ||
  CASE WHEN IngredientName IS NULL THEN " is vegetarian"
  ELSE " is not vegetarian because it contains "
       || IngredientName END || ".") AS announcement
FROM Recipes LEFT NATURAL JOIN
(SELECT * FROM Recipe_Ingredients
  LEFT JOIN Ingredients ON
  Recipe_Ingredients.IngredientID=Ingredients.IngredientID
  WHERE IngredientClassID IN (2,10))
GROUP BY RecipeTitle;
```

What would happen if we instead do:

```
GROUP BY RecipeID;
```

Group String Concatenation Aggregator

Remember the `GROUP BY` rules:

- `SELECT` can only use columns present in `GROUP BY`
- Any other columns in `SELECT` can only be in aggregators

`GROUP_CONCAT(X, Y)` → returns a string
concatenates all non-NULL values of `X`
optional field separator `Y`, defaults to “ , ”

PostgreSQL: `STRING_AGG ( expression, separator [order_by_clause] )`

Query without duplication – the right version!

```
SELECT (RecipeTitle ||
  CASE WHEN GROUP_CONCAT(IngredientName) IS NULL THEN " is vegetarian"
  ELSE " is not vegetarian because it contains "
       || GROUP_CONCAT(IngredientName) END || ".") AS announcement
FROM Recipes LEFT NATURAL JOIN
(SELECT * FROM Recipe_Ingredients
  LEFT JOIN Ingredients ON
  Recipe_Ingredients.IngredientID=Ingredients.IngredientID
  WHERE IngredientClassID IN (2,10))
GROUP BY RecipeTitle;
```

GROUP_CONCAT( )

…

9 Pollo Picoso is not vegetarian because it contains Chicken Leg,Chicken Thigh.

…

Regular Expressions (REGEXP)

- Regular Expressions are patterns that match text

```
… WHERE column REGEXP "pattern" …
```

- They are much more flexible than the LIKE expressions we have used
  - LIKE expressions use % to represent a sequence of unknown characters and _ to represent a single unknown character
- Regular Expressions can be much more specific:
  - Match different types of characters (letters, numbers, whitespace)
  - Allows sub-patterns to repeat
  - … and more
- SQLite, MySQL, and every major DBMS support REGEXP, although the syntax details may vary
- Regular Expressions are also used in many other programing languages and in the grep command-line tool on Mac and Unix

A simple Regular Expression: `barf`

Matches:

- barf
- barfly
- I embarfed on my journey.
- I barfed at McDonalds.

Does *not* match:

- Barf
- BARF
- This bar finally closed.
- I enjoyed my meal at McDonalds.
- “arf!” – “Good boy”, he said.

Beginning and end of the text

Normally, regular expressions match anywhere in the text, but we can change that behavior as follows:

`^` matches the beginning of the text
`$` matches the end of the text

`^Hello` matches “Hello World.” but does not match “Big Hello”

`world$` matches “hello world” but does not match “world cup”

`^hello world$` matches “hello world” and nothing else

Sets of characters

`.` (period) matches any one character (as does `_` with `LIKE` expressions)

Square braces `[…]` specify a set of characters, any of which can match

`[aA]` specifies by inclusion: either “a” or “A”

`[a-z]` specifies by range: any of the characters between “a” and “z”

`[^b]` specifies by exclusion: any character *other than* “b”

These sets can be combined, as follows:

`[a-zA-Z01]` specifies any English letter or the numbers 0 or 1

`[^CDA]` specifies any character other than “C” “D” or “A”

Repetition

`*` lets the previous thing repeat $k \geq 0$ times

`+` lets the previous thing repeat $k \geq 1$ times

`{n,m}` lets the previous thing repeat k times, $n \leq k \leq m$

`?` lets the previous thing be optional (shorthand for {0,1})

Logical OR

`(this|that)`

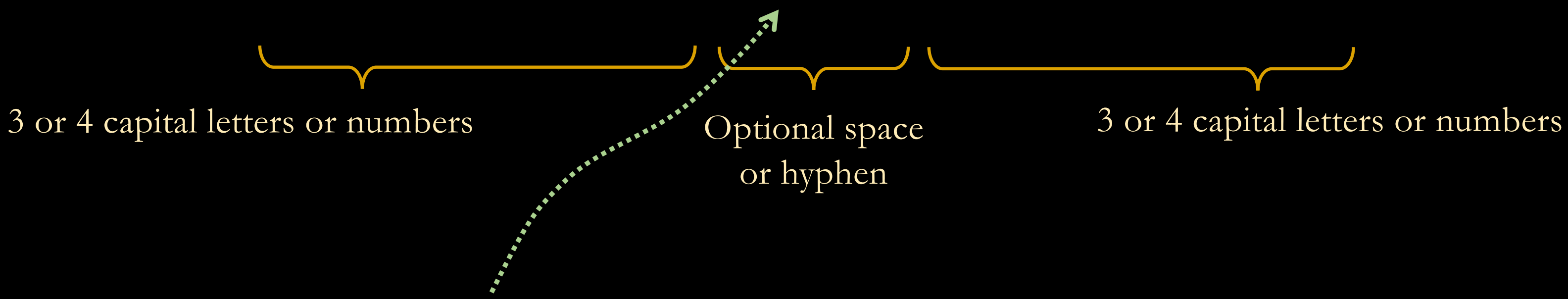
`.*` matches anything because it matches any one character repeated any number of times

Car license plate example

Let’s say we want to match text that could be car license plates

- Must be 6 to 8 characters (capital English letters or numbers), optionally with an additional space or dash in the middle
- e.g., “123-AB3” or “4FDK930”

```
[A-Z0-9]{3,4}[ \-]?[A-Z0-9]{3,4}
```



3 or 4 capital letters or numbers

Optional space or hyphen

3 or 4 capital letters or numbers

“\” is needed to “escape” the normal meaning of hyphen inside square brackets. We want the literal hyphen character; we are not specifying a range of characters.