CSC 365

Introduction to Database Systems

SQL / Relational Algebra

CAL POLY

SQL	Relational Algebra Operator	
`SELECT DISTINCT`	π	Projection
`FROM`	$\times$	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

Syntax:

```
SELECT <list of columns>
FROM <table(s)>
[ INNER|CROSS|LEFT|RIGHT|FULL JOIN <table> ON <join condition> ]
[ WHERE <predicate> ]
[ GROUP BY <columns> ]
[ HAVING <group filter> ]
[ ORDER BY <column(s)>
```

SQL - Scalar Functions

CAL POLY

We have seen **scalar** functions in SQL that operate on a single row and return just one value. Previous examples:

```
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
```

SQL SELECT - Aggregate Functions

CAL POLY

In addition to scalar functions, SQL SELECT also supports **Aggregate Functions,** which summarize multiple rows.

- `COUNT()`
- `MIN()`
- `MAX()`
- `SUM()`
- `AVG()`

COUNT()

Simple examples:

```
SELECT COUNT(*) FROM Student      -- total number of students
```

With a `WHERE` clause:

```
SELECT COUNT(*) FROM Student      -- number of CSC majors
WHERE MajorCode = 'CSC'
```

MIN()

Returns the lowest value from a single column.

```
SELECT MIN(DateEnrolled) FROM Student
```

May also be combined with WHERE:

```
SELECT MIN(DateEnrolled) FROM Student
WHERE MajorCode IN ('CSC', 'SE', 'CPE')
```

MAX()

`MAX()` returns the highest value from a single column.

```
SELECT MAX(MaxSpeed)
FROM Airplane
```

It's valid to include multiple aggregates in a single SELECT:

```
SELECT MAX(MaxSpeed), MIN(MaxSpeed), COUNT(*)
FROM Airplane
```

SUM() / AVG()

CAL POLY

`SUM()` and `AVG()` operate on numeric (integer or decimal) columns, returning the arithmetic sum or average.

```
SELECT SUM(MaxSpeed), COUNT(*), AVG(MaxSpeed)
FROM Airplane
```

Handling `NULL` values:

```
SELECT SUM(MaxSpeed) / COUNT(*) AS CalcAvg, AVG(MaxSpeed)
FROM Airplane
```

Aggregate Functions: ALL vs DISTINCT

By default, SQL aggregate functions operate in "ALL" mode:

```
SELECT COUNT(Make), COUNT(ALL Make) FROM Airplane
```

`DISTINCT` causes the aggregate function to consider only unique values:

```
SELECT COUNT(Make), COUNT(DISTINCT Make) FROM Airplane
```

The functions `AVG()`, `COUNT()`, and `SUM()` support `DISTINCT` mode

SQL SELECT - Aggregate Functions

- Used alone, an aggregate function collapses all rows into *one* summarized row.
- What if we want to include other column values in our result set?

```
SELECT Make, AVG(MaxSpeed)
FROM Airplane
```

GROUP BY

CAL POLY

The `GROUP BY` clause, along with aggregate functions, allows us to identify which column(s) to use as the basis for our summary:

```
SELECT Make, AVG(MaxSpeed)
FROM Airplane
GROUP BY Make
```

```
SELECT DAYNAME(DateEnrolled), COUNT(*)
FROM Student
GROUP BY DAYOFWEEK(DateEnrolled), DAYNAME(DateEnrolled)
ORDER BY DAYOFWEEK(DateEnrolled)
```

`DAYNAME()` and `DAYOFWEEK()` are *MySQL-specific* scalar date functions. Full list here

GROUP BY

`GROUP BY` is permitted on both columns and the result of *scalar* functions:

```
-- total guest count for each non-rustic room / month
SELECT RoomName, YEAR(CheckIn), MONTH(CheckIn),
  SUM(Adults + Kids) AS TotalGuests
FROM INN.rooms AS rm
  INNER JOIN INN.reservations AS res ON rm.RoomCode = res.Room
WHERE rm.decor NOT IN ('rustic')
GROUP BY RoomName, YEAR(CheckIn), MONTH(CheckIn)
```

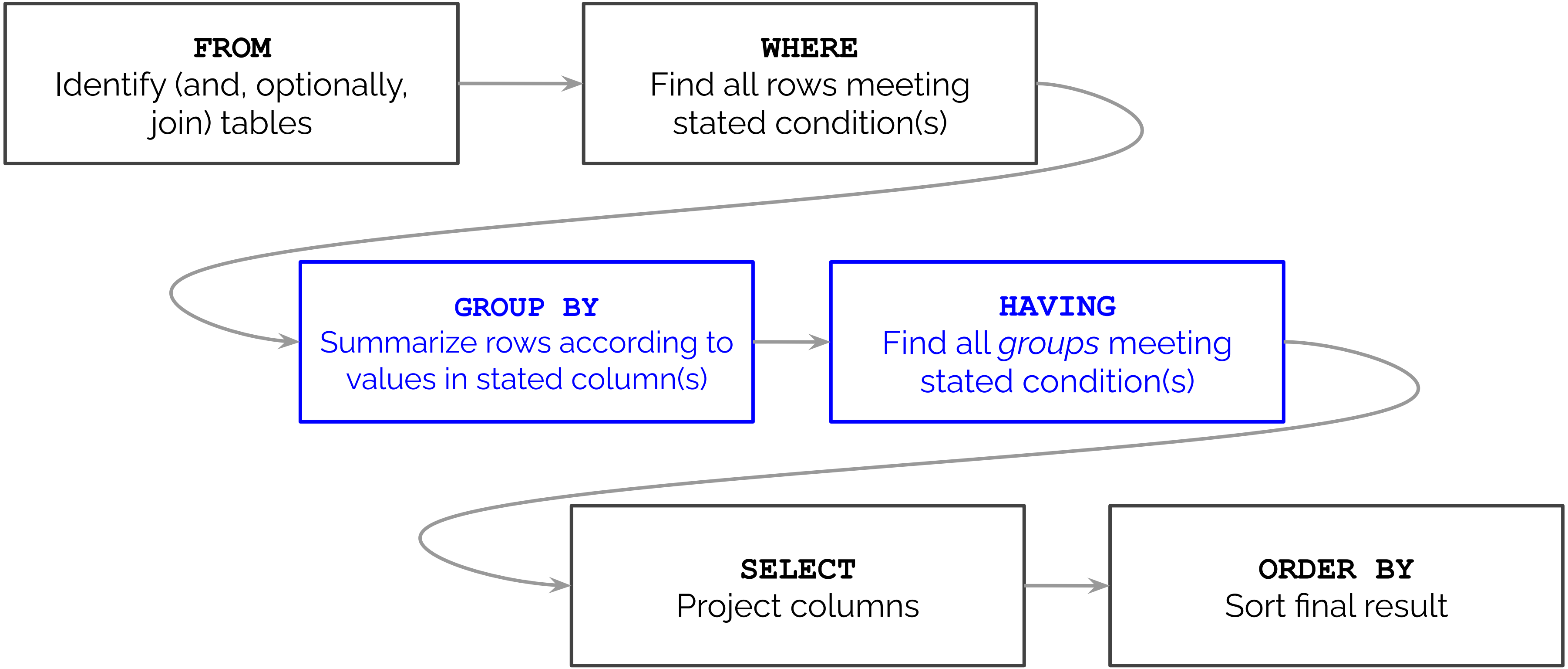
GROUP BY with Conditions

Conditions on aggregate values require special treatment.

List all dates during October 2010 on which more than 5 people checked in to our inn. First try:

```
SELECT CheckIn, SUM(Adults + Kids) AS TotalGuests
FROM INN.reservations
WHERE CheckIn BETWEEN '2010-10-01' AND '2010-10-31'
  AND SUM(Adults + Kids) > 5
GROUP BY CheckIn
ORDER BY CheckIn
```

SQL SELECT Logical Processing Order



SQL SELECT Logical Processing Order

Order	Clause
5	SELECT
6	DISTINCT
1	FROM … JOIN … ON
2	WHERE
3	**GROUP BY**
4	**HAVING**
7	UNION
8	ORDER BY

GROUP BY with Conditions

CAL POLY

The `HAVING` clause, used along with `GROUP BY`, allows us to apply conditions that involve aggregate functions.

List all dates during October 2010 on which more than 5 people checked in to our inn.

```
SELECT CheckIn, SUM(Adults + Kids) AS TotalGuests
FROM INN.reservations
WHERE CheckIn BETWEEN '2010-10-01' AND '2010-10-31'
GROUP BY CheckIn
HAVING SUM(Adults + Kids) > 5
ORDER BY CheckIn
```

AIRLINES-6 - GROUP BY

Find all airports that are the source of at least three Southwest flights. Report just the three-letter codes of the airports each code exactly once, in alphabetical order.

```
SELECT f.Source
FROM flights f
  INNER JOIN airlines a ON f.Airline = a.Id
WHERE a.Abbr = 'Southwest'
GROUP BY f.Source
HAVING COUNT(*) >= 3
ORDER BY f.Source
```

GROUP BY Examples (BAKERY)

1. Find the average price of a Cookie.

2. List all customers who have spent more than $100 at the bakery. Include the customer's total spending and the number of days between the customer's most recent purchase and the end of October 2007.

3. What was the *average receipt amount* on Fridays versus Mondays?

4. List first and last name(s) of customers who have purchased *every* type of Danish

GROUP BY in Relational Algebra

CAL POLY

The *grouping operator* γ combines the effect of grouping and aggregation.

$$\gamma_L(R)$$

L is a list that may contain any of the following:

- One or more attributes of R, which will serve as *grouping attributes*
- Aggregation function(s) with an arrow and new name for the aggregate value

GROUP BY in Relational Algebra

CAL POLY

Examples of the *grouping operator* γ in relational algebra

$\gamma_{\text{Make, MAX(MaxSpeed)} \rightarrow \text{SpeediestInFleet}}(\text{AIRPLANE})$

```
-- highest maximum speed for each airplane maker
SELECT Make, MAX(MaxSpeed) AS SpeediestInFleet
FROM Airplane
GROUP BY Make
```

$\gamma_{\text{CheckInDate, SUM(Adults + Kids)} \rightarrow \text{TotalGuests}}(\sigma_{\text{CheckIn >= '2010-10-02' } \wedge \text{ CheckIn <= '2010-10-31'}}(\text{RESERVATIONS}))$

```
-- number of guests who checked in on each date
SELECT CheckIn, SUM(Adults + Kids) AS TotalGuests
FROM reservations
WHERE CheckIn BETWEEN '2010-10-20' AND '2010-10-31'
GROUP BY CheckIn
```

GROUP BY - Alternate Notation

Some references use a calligraphic G ($\boldsymbol{\mathcal{G}}$) rather than lowercase gamma (γ):

$${}_{\text{Make}}\mathcal{G}_{\text{MAX(MaxSpeed)} \rightarrow \text{SpeediestInFleet}}(\text{AIRPLANE})$$

$${}_{\text{CheckInDate}}\mathcal{G}_{\text{SUM(Adults + Kids)} \rightarrow \text{TotalGuests}}(\sigma_{\text{CheckIn >= '10/1/2010'}}(\text{RESERVATIONS}))$$

On the topic of Relational Algebra...

Two more "extended" relational algebra operators:

- The duplicate-elimination operator **δ** turns a bag into a set by eliminating all but one copy of each tuple (in SQL: `SELECT DISTINCT *...`)

- The sorting operator **τ** turns a relation into a *list* of tuples, sorted according to one or more attributes *(Sorting should be applied cautiously. Some relational algebra operators do not make sense on lists.)*

GROUP BY / HAVING in Relational Algebra

CAL POLY

How would we represent the following query in relational algebra, applying our extended relational algebra operators, gamma (γ) and tao (τ)?

```
-- dates during October 2010 on which more than 5 people checked in
SELECT CheckIn, SUM(Adults + Kids) AS TotalGuests
FROM INN.reservations res
  JOIN INN.rooms r ON r.RoomCode = res.Room
WHERE CheckIn BETWEEN '2010-10-01' AND '2010-10-31'
GROUP BY CheckIn
HAVING TotalGuests > 5
ORDER BY CheckIn
```

Can we do this in a single SQL statement?

BAKERY: List total sales amount for each food and flavor combination, along with the percent of sales for each flavor within its type of pastry (Twist, Meringue, Cake, Danish, etc.)

Flavor	Food	TotalSales	PctOfFoodSales
Almond	Twist	20.70	100
Chocolate	Meringue	25	60.8
Vanilla	Meringue	16.10	39.2
...	...	...	...

% should sum to 100 for each pastry type (Food column)

Window Functions

Window functions allow us to perform calculations across a result set. Somewhat like aggregation / `GROUP BY`, but window functions *do not* collapse rows. All rows are preserved.

Syntax (in `SELECT` clause):

Rows with the same value for <column list> fall into the same partition

```
<window function> OVER (PARTITION BY <column list> [ORDER BY <colums>])
```

Available window functions include familiar aggregate functions
(SUM, MIN, MAX, COUNT, AVG)
as well as a few new functions:
RANK, ROW_NUMBER, NTILE, LAG, LEAD

Window Functions

CAL POLY

Function Name	Description
RANK()	Rank of current row within its partition, *with gaps*
ROW_NUMBER()	Number of current row within its partition
DENSE_RANK()	Rank of current row within its partition, without gaps
FIRST_VALUE()	Value of argument from first row of partition
LAST_VALUE()	Value of argument from last row of partition
NTH_VALUE()	Value of argument from N-th row of window frame
NTILE()	Bucket number of current row within its partition.

(plus a few more…)

Window Function Examples

```
-- All employees, along with department average salary
SELECT DeptID, EmpID, AnnualSalary,
  AVG(AnnualSalary) OVER (PARTITION BY DeptID) as DeptAverage
FROM employee;

-- In which order were employees hired within their department?
SELECT DeptID, EmpID, HireDate,
  RANK() OVER (PARTITION BY DeptID ORDER BY HireDate) as HireOrder
FROM employee;


-- RANK() vs DENSE_RANK()
```

Named Windows

Shared, named windows can be defined using the `WINDOW` clause, which appears between the `HAVING` and `ORDER BY` clauses.

```
WINDOW <window name> AS (window spec)
    [, <window name> AS (window spec)] ...
```

Named Window Example

CAL POLY

```
SELECT DeptID, EmpID, HireDate,
  RANK() OVER w AS HireOrder,
  DENSE_RANK() OVER w AS DenseHireOrder,
  FIRST_VALUE(HireDate) OVER w AS EarliestDeptHire
FROM employee
WINDOW w AS (PARTITION BY DeptID ORDER BY HireDate)
```

Named Window Example 2


```
-- How much less is each employee paid than the highest compensated
-- in his/her department
SELECT DeptID, EmpID, HireDate, AnnualSalary,
  RANK() OVER (w ORDER BY HireDate) AS HireOrder,
  AVG(AnnualSalary) OVER w AS DeptAvgSalary,
  MAX(AnnualSalary) OVER w AS DeptMaxSalary,
   (MAX(AnnualSalary) OVER w) - AnnualSalary AS SalaryDiff
FROM employee
WINDOW w AS (PARTITION BY DeptID)
```

Window Functions Applied

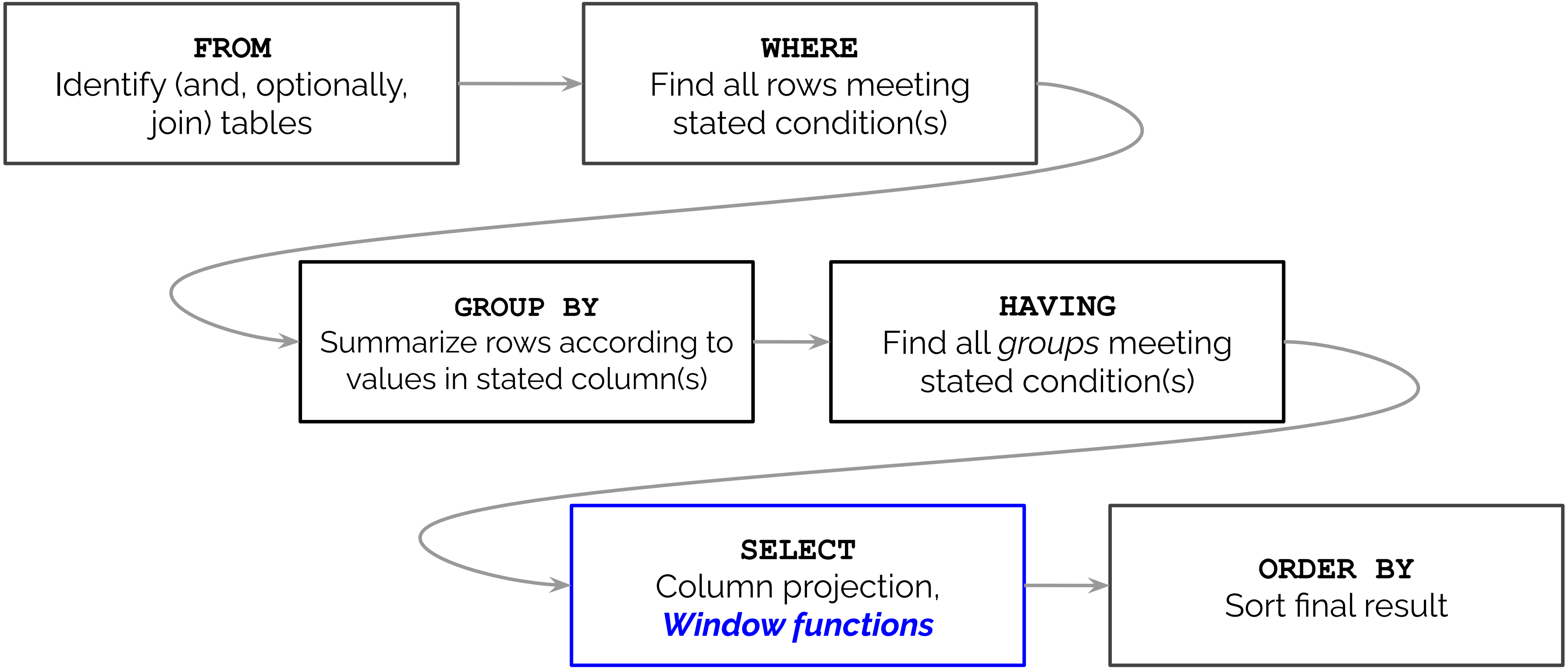
BAKERY: List total sales amount for each food and flavor combination, along with the percent of sales for each flavor within its type of pastry (Twist, Meringue, Cake, Danish, etc.)

Flavor	Food	TotalSales	PctOfFoodSales
Almond	Twist	20.70	100
Chocolate	Meringue	25	60.8
Vanilla	Meringue	16.10	39.2
...	...	...	...

% should sum to 100 for each pastry type (a.k.a. food)

SQL SELECT Logical Processing Order

CAL POLY



Window Function Summary

CAL POLY

- Window functions may be used only in the `SELECT` and `ORDER BY` clauses (recall logical query processing order)
- Window functions execute *after* `GROUP BY` and aggregate functions.
    - Valid to use an aggregate function call in the arguments of a window function
    - Not possible to use window functions within regular aggregate functions
- Not supported by all RDBMSs / versions
    - Available only since MySQL v8.0
    - No support in SQLite

OLTP / OLAP

Cal Poly

Online Transaction Processing (OLTP): emphasis on high throughput, inserts/updates, speed, concurrency, and recoverability.

Online Analytical Processing (OLAP): complex queries and analysis (often involving aggregation), sometimes referred to as decision-support queries.

OLAP / OLTP: Sample SQL

CAL POLY

OLTP	OLAP
INSERT INTO employee (EmpID, DeptID, Salary, FirstName, LastName, HireDate) VALUES (...); SELECT * FROM employee WHERE HireDate > '2018-01-01' AND DeptID IN ('ENG', 'HR', 'ACCTG'); UPDATE employee SET Salary = Salary * 1.1 WHERE HireDate BETWEEN '2017-01-01' AND '2017-03-31' AND Salary < 50000;	SELECT AVG(Salary), MIN(Salary), MAX(Salary) FROM employee; SELECT DeptID, COUNT(EmpID) FROM employee WHERE DeptID <> 'SALES' GROUP BY DeptID HAVING AVG(Salary) > 50000;

Low *selectivity* (few rows accessed at a time)
high *projectivity* (number of columns accessed)

Low *projectivity*, high *selectivity*

OLAP / OLTP Storage Strategies

CAL POLY

Online Analytical Processing (OLAP) often benefits from column-oriented storage: all values in a particular column are stored contiguously.

EmpID	Department	Salary
001	HR	60000
002	ENG	80000
003	ENG	85000
004	SALES	120000

Online Transaction Processing (OLTP) typically uses row storage: data for each tuple/record stored together.

001	HR	60000
001	ENG	80000
001	ENG	85000
001	SALES	120000