SQL: Aggregation, Joins, and Triggers

CSC 343
Winter 2021

MICHAEL LIUT (MICHAEL.LIUT@UTORONTO.CA)
ILIR DEMA (ILIR.DEMA@UTORONTO.CA)

DEPARTMENT OF MATHEMATICAL AND COMPUTATIONAL SCIENCES
UNIVERSITY OF TORONTO MISSISSAUGA

Aggregation Operators

Column values are aggregated, and a single value is returned.

SUM, AVG, COUNT, MIN, and MAX.

These operations can be applied on a SELECT clause in a query to produce the aggregation on the column.

e.g. COUNT(*) → counts the number of tuples in a table.

Aggregation Operations are Column Operations!

Example: Aggregation

From Sells(bar, beer, price), find the average price of Bud.

```
SELECT AVG (price)
FROM   Sells
WHERE  beer = 'Bud';
```

Aggregation Operators

An aggregation operator may not appear in the WHERE clause unless it is in a subquery contained in a HAVING clause or a SELECT list, and the column being aggregated is an outer reference.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

An example will shortly follow.

Duplicates in an Aggregation

To eliminate duplicates use DISTINCT inside an aggregation.

Assignment Project Exam Help

Example: Find the number of *different* prices charged for Bud.

https://powcoder.com

SELECT   COUNT (DISTINCT price)

Add WeChat powcoder

FROM   Sells

WHERE beer = 'Bud';

NULL Values in Aggregation

NULL values are **NEVER** contributed to a SUM, AVG, or COUNT.

NULL values can **NEVER** be the MIN or MAX of a column.

However, if all the values in the column are NULL, then the result of the aggregation is also NULL.

Exception: The COUNT of an empty set is 0.

Example: The NULL Effect

```
SELECT   COUNT(*)
FROM     Sells
WHERE    beer = 'Bud';
```

The number of bars that sell Bud.

```
SELECT   COUNT(price)
FROM     Sells
WHERE    beer = 'Bud';
```

The number of bars that sell Bud at a known price.
i.e. where the price is NOT NULL.

Recall: Sells(bar, beer, price)

Example: A Simple Query

Find the age of the youngest employee at each rating level.

SELECT MIN(age)

FROM   Employees

WHERE rating = i;

Note: "i" represents a rating value.

Grouping

We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.

The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

> **SELECT** rating, MIN(age)
> **FROM**   Employees
> **GROUP BY** rating;

Example: Grouping

Find the average price for each beer from Sells(bar, beer, price)

SELECT   beer, AVG(price)

FROM   Sells

GROUP BY beer;

Result:

beer	AVG(price)
Bud	2.33
Miller	4.55
...	...

SELECT clause Restrictions with Aggregation

To continue with the paragraph on slide #4:

- If any aggregation is used, then each element of the SELECT list must be either:
    1. Aggregated, or
    2. An attribute on the GROUP BY list

Example: Grouping

For each drinker, find the average price of Bud at the bars they frequent.
You will use: Sells(bar, beer, price)   Frequents(drinker, bar)

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Compute all drinker-bar-price triples for 'Bud'.

SELECT drinker, AVG(price)

FROM   Frequents, Sells

WHERE beer = 'Bud' AND Frequents.bar = Sells.bar

GROUP BY drinker;

Then group them by drinker.

Illegal Query Example

```
SELECT   bar, beer, MIN(price)
FROM    Sells
GROUP BY  bar;
```

This query is illegal in SQL.

There is only one tuple output for each bar. Thus, no unique way to select which beer to output

Illegal Query Example In Detail

SELECT   bar, beer, MIN(price) AS minPrice
FROM   Sells
GROUP BY bar;

Result:

bar	beer	minPrice
Joe	?	3.00
Tom	?	3.50
Jane	?	3.25

{Bud, Miller, Coors}?

Ideally, we would want to GROUP BY beer.

Sells:

bar	beer	price
Joe	Bud	3.00
Joe	Miller	4.00
Tom	Bud	3.50
Tom	Miller	4.25
Jane	Bud	3.25
Jane	Miller	4.75
Jane	Coors	4.00

Only one tuple output for each bar, no unique way to select which beer to output.

HAVING Clauses

HAVING <condition> may follow a GROUP BY clause.

If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

Example: Having

From Sells(bar, beer, price) and Beers(name, manf) find the average price of the beers that are either served in at least three bars or are manufactures by Pete's.

SELECT   beer, AVG(price)
FROM   Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR
beer IN (SELECT   name
           FROM   Beers
           WHERE   manf = 'Pete''s'
);

Beer groups with at least 3 non-NULL bars.

Beers manufactured by Pete's.

Requirements of HAVING Conditions

Anything goes in a moment.

Outside subqueries, they may refer to attributes only if they are either:

1. A grouping attribute, or

2. Aggregated.

(same condition as in SELECT clauses with aggregation)

Aggregation Operators

An aggregation operator may not appear in the WHERE clause. It must be contained in a HAVING clause or a SELECT, and the column being aggregated is an outer reference.

SELECT empID, SUM(amount)
FROM   Sales
GROUP BY empID
HAVING SUM(amount) > 20000;

OR

SELECT empID, SUM(amount)
FROM   Sales
WHERE empID **IN** (
    SELECT MAX(empID)
    FROM Employees)
GROUP BY empID;

A Final Example

SELECT bar, SUM(qty) AS sumQ

FROM Sells

GROUP BY bar

HAVING SUM(qty) > 4;

Sells:

bar	beer	price	qty
Joe	Bud	3.00	2
Joe	Miller	4.00	2
Tom	Bud	3.50	1
Tom	Miller	4.25	4
Jane	Bud	3.25	1
Jane	Miller	4.75	3
Jane	Coors	4.00	2

Result:

bar	sumQ
Tom	5
Jane	6

Cross Product

A.K.A. Cartesian Product. Denoted: ⨯

Evaluating joins involves combining two or more relations.

Given two relations S and R, each row of S is paired with each row of R.

Resulting Schema: one attribute from each attribute of S and R.

Example: Cross Product

Sells:

bar	beer	price
Joe	Bud	3.00
Tom	Miller	4.00
Jane	Lite	3.25

Frequents:

drinker	bar
Aaron	Joe
Mary	Jane

Result:

drinker
Aaron

Sells X Frequents

(bar)	beer	price	drinker	(bar)
Joe	Bud	3.00	Aaron	Joe
Joe	Bud	3.00	Mary	Jane
Tom	Miller	4.00	Aaron	Joe
Tom	Miller	4.00	Mary	Jane
Jane	Lite	3.25	Aaron	Joe
Jane	Lite	3.25	Mary	Jane

SELECT   drinker
FROM   Frequents, Sells
WHERE beer = 'Bud' AND Frequents.bar = Sells.bar;

Cross Product

In general: Assignment Project Exam Help

https://powcoder.com

	Table A	Table B	A X B
Rows	N	M	N * M
Columns	C	K	C + K

Add WeChat powcoder

Example: Cross product of 2x2-table **A** and 2x3-table **B**.

| A{1,1} | A{1,2} |
| A{2,1} | A{2,2} |

X

| B{1,1} | B{1,2} | B{1,3} |
| B{2,1} | B{2,2} | B{2,3} |

A{1,1}	A{1,2}	B{1,1}	B{1,2}	B{1,3}
A{1,1}	A{1,2}	B{2,1}	B{2,2}	B{2,3}
A{2,1}	A{2,2}	B{1,1}	B{1,2}	B{1,3}
A{2,1}	A{2,2}	B{2,1}	B{2,2}	B{2,3}

Joined Relations

Join operations take two relations and returns as a result another relation.

A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join.

Example: Join Operations

Course:

course_id	title	department	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp Sci	4
CS-315	Robotics	Comp Sci	3

Prereq:

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Observe:
1. The prerequisite information is missing for CS-315.
2. The course information is missing for CS-347.

Types of Joins
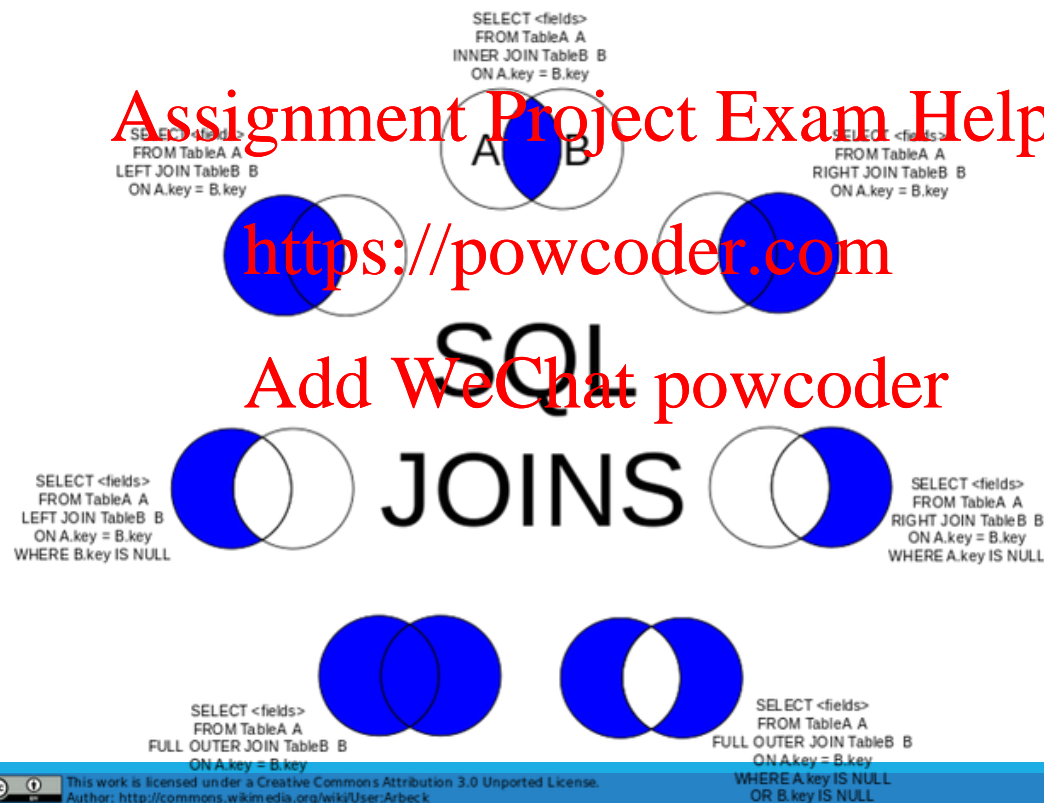
SELECT <fields>
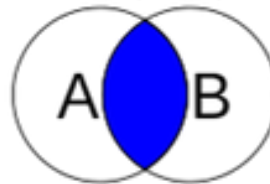FROM TableA A
INNER JOIN TableB B
ON A.key = B.key

SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key

SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key

SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
WHERE B.key IS NULL

SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL

SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key

SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL

SQL JOINS

Assignment Project Exam Help
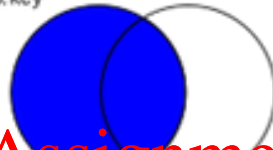
https://powcoder.com

Add WeChat powcoder

SELECT <fields>
FROM TableA A
INNER JOIN TableB B
ON A.key = B.key

SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key

A B

SELECT <fields>
FROM TableA A
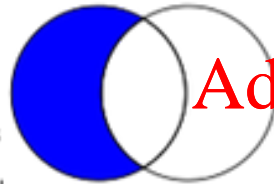RIGHT JOIN TableB B
ON A.key = B.key

SQL
JOINS
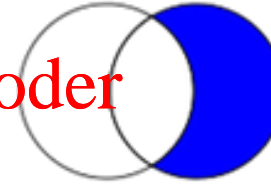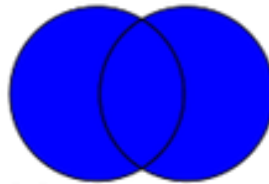
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
WHERE B.key IS NULL
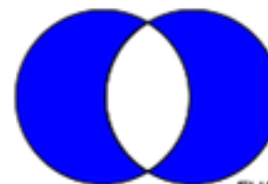
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL

SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key

SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL

Outer Join

An extension of the join operation that avoids loss of information.

Suppose you have two relations *R* and *S*. A tuple of *R* that has no tuple of *S* with which it joins is said to be dangling.

- Similarly for a tuple of *S*.

Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.

Outerjoin preserves dangling tuples by padding them with NULL.

Left Outer Join

Course:

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp Sci	4
CS-315	Robotics	Comp Sci	3

Prereq:

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Result:

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp Sci	4	CS-101
CS-315	Robotics	Comp Sci	3	NULL

Right Outer Join

Course:

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp Sci	4
CS-315	Robotics	Comp Sci	3

Prereq:

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Result:

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp Sci	4	CS-101
CS-347	NULL	NULL	NULL	CS-101

Full Outer Join

Course:

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp Sci	4
CS-315	Robotics	Comp Sci	3

Prereq:

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Result:

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp Sci	4	CS-101
CS-315	Robotics	Comp Sci	3	NULL
CS-347	NULL	NULL	NULL	CS-101

Inner Join

Course:

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp Sci	4
CS-315	Robotics	Comp Sci	3

Prereq:

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Course **INNER JOIN** Prereq **ON**
Course.course_id = Prereq.course_id

Result:

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp Sci	4	CS-101

Outer joins

R OUTER JOIN *S* is the core of an outer join expression.

Modified by the following:

1. Optional NATURAL in front of OUTER.
   - Check equality on all common attributes.
   - No two attributes with the same name in the output.
2. Optional ON <condition> after JOIN.
3. Optional LEFT, RIGHT, or FULL in front of OUTER.
   - LEFT = pad dangling tuples of *R* only.
   - RIGHT = pad dangling tuples of *S* only.
   - FULL = pad both; this is the DEFAULT choice.

Example: Outer Join

R:

A	B
1	2
4	5

S:

B	C
2	3
6	7

R NATURAL FULL OUTER JOINS

Result:

A	B	C
1	2	3
4	5	NULL
NULL	6	7

NOTE: (1,2) joins with (2,3), but the other two tuples are dangling!

Triggers

A procedure that is automatically invoked by the DBMS in response to specified changes to the database.

- ◦ Typically invoked by the DBA.

A database that has a set of associated triggers is called an *active database*.

Triggers

A trigger description contains three parts:

1. Event: a change to the database that activates the trigger.
2. Condition: a query or test that is run when the trigger is activated.
3. Action: a procedure that is executed when the trigger is activated and its condition is true.

Triggers

A trigger can be thought of as a 'passport' which monitors a database, and is executed when the database is modified in a way that matches the event specifications.

  i.e. INSERT, UPDATE, and DELETE statements can activate a trigger.

Users are often unaware that a trigger was executes as a side effect of their program.

Daemon

No, not the ancient Greek word that refers to benevolent or benign natural spirits... I am talking about them in the computing sense! ☺

A computer program that runs as a background process instead of being controlled by an interactive user.

e.g. *SSHD* is the daemon that serves incoming *SSH* connections.

This is seen in computer operating systems, specifically in multitasking.

In UNIX, the parent process of a daemon is usually the *init* process.

Now back to Triggers...

Triggers

A *condition* in a trigger can be a Boolean expression or a query.

e.g. All employees salaries are less than $100,000.

A query is interpreted as TRUE if the answer set is non-empty.
- This invokes the action associated with the trigger.

A query is interpreted as FALSE if it has no answers.
- No action is invoked.

Important Issues Associated With Triggers

When a trigger action executes in a relation, we must specify at what point in the sequence of events it occurs.

- ◦ BE CAREFUL!

Depending on what the trigger does, we may want to modify if its action occurs *before* or *after* changes are made to relations.

Important Issues Associated With Triggers

Example: let's say that we have a statement that INSERTS records into a Student's table.

◦ This activates a trigger that maintains statistics on how many students younger than 18 years of age are inserted at one time.

Does this trigger happen *before* or *after* the change?

Important Issues Associated With Triggers

Does this trigger happen *before* or *after* the change?

You have different options, for example:

1. A trigger that initializes a variable used to count the number of qualifying insertions should be executed *before*.

2. A trigger that executes once *after* each record is inserted.
   ◦ Maybe we require the values in the new record to determine the action.

Example: Triggers

Let's say that we have a statement that INSERTs records into a Student's table.

- ◦ This activates a trigger that maintains statistics on how many students younger than 18 years of age are inserted at one time.

Now let's assume that we must examine the age attribute of the Student table to decide whether to increment the count.

- ◦ The triggering event here would occur for each modified record.

Example: Triggers

```
CREATE  TRIGGER  init_count  BEFORE INSERT ON Students        /* Event */
        DECLARE
                count  INTEGER;
        BEGIN
                count := 0;                                  /* Action */
        END
```

"new" is just an inserted tuple.

Note: The init_count trigger is executed once per INSERT, thus FOR EACH ROW was omitted, however, it would be considered a **statement-level** trigger.

```
CREATE  TRIGGER  incr_count  AFTER INSERT ON Students        /* Event */
        WHEN  (new.age < 18)                                 /* Condition */
        FOR EACH ROW
        BEGIN                                                /* Action */
                count := count + 1;
        END
```

Called a **row-level** trigger.

Row-Level Trigger vs. Statement-Level Trigger

A statement-level trigger is activated once.

A row-level trigger is activated once per iteration of the loop.

e.g.   UPDATE attribute

SET columnOne = columnOne +1;

The statement-level trigger will be activated one time (even if no rows are updated).

The row-level trigger will be activated millions of times (dependent on the size/iters).

Important Note on Triggers

Watch out for chain activations!

Depending on your execution of the action, you could potentially activate another trigger. You could potentially even activate the same trigger.
- These are called **recursive triggers**

Recursive triggers are mostly unpredictable because of the combination of chain activations and order which the DBMS processes this sequence can be difficult to understand.

Constraints vs. Triggers

Triggers are commonly used to maintain database consistency.
- Must ensure that an Integrity Constraint is not more properly suited.

Constraints are easier to understand as they are not operationally defined.

Constraints prevent inconsistency by any kind of statement.

Constraints afford the opportunity for the DMBS to optimize.

Triggers allow us to impose general (flexible) constraints

Triggers alert users to unusual events (reflected in updates to the database).

Triggers can generate a log of events to support auditing and security checks.

Trigger Flexibility

Suppose that we have a table called Orders with fields *iternid*, *quantity*, *customerid*, and *unitprice*. When a customer places an order, the first three field values are filled in by the user (in this example, a sales clerk). The fourth field's value can be obtained from a table called Items, but it is important to include it in the Orders table to have a complete record of the order, in case the price of the item is subsequently changed. We can define a trigger to look up this value and include it in the fourth field of a newly inserted record. In addition to reducing the number of fields that the clerk has to type in, this trigger eliminates the possibility of an entry error leading to an inconsistent price in the Orders table.

Trigger Flexibility

Continuing with this example, we may want to perform some additional actions when an order is received. For example, if the purchase is being charged to a credit line issued by the company, we may want to check whether the total cost of the purchase is within the current credit limit. We can use a trigger to do the check; indeed, we can even use a CHECK constraint. Using a trigger, however, allows us to implement more sophisticated policies for dealing with purchases that exceed a credit limit. For instance, we may allow purchases that exceed the limit by no more than 10% if the customer has dealt with the company for at least a year, and add the customer to a table of candidates for credit limit increases.

Alert to Unusual Events

We may want to check if a customer placing an order has made enough purchases in the past month to qualify for an additional discount.

- How? Implement a trigger to check recent purchases and displays message.

- IF TRUE: Inform the Sales Clerk.

- Purpose? To UPSELL or XSELL additional products as they are receiving a discount.

Some Review Questions

1. What are nested queries? How and when would you use the operators IN, EXISTS, UNIQUE, ANY, and ALL?

2. What are NULL values? Are they supported in the relational model? How do they affect the meaning of queries? Can primary key fields of a table contain NULL values?

3. What is a trigger? What are its 3 parts? What is the difference between a row-level and statement-level trigger?

4. What is grouping? What is its interaction of the HAVING and WHERE clauses? Mention any restrictions that must be satisfies by the fields which appear in the GROUP BY clause.

Questions?

Q & A

THANKS FOR LISTENING

I'LL BE ANSWERING QUESTIONS NOW

UNIVERSITY OF
TORONTO
MISSISSAUGA

Citations, Images and Resources

Database Management Systems (3rd Ed.), Ramakrishnan & Gehrke

Some content is based off the slides of Dr. Fei Chiang – http://www.cas.mcmaster.ca/~fchiang/

http://csharpcorner.mindcrackerinc.netdna-cdn.com/UploadFile/BlogImages/06112016031910AM/sql.png

https://lukaseder.files.wordpress.com/2015/10/venn.png?w=662

http://stackoverflow.com/questions/6319183/aggregate-function-in-sql-where-clause

http://www.sql-server-performance.com/2007/aggregate-may-not-appear-where-clause/

http://bioinfo.mbb.yale.edu/course/projects/talk-1/db07.html