Data 101/CS 187: Data Engineering Fall 2026 Discussion 2 – September 11th, 2026

Relational Algebra, Subqueries, CTEs

Instructor: Aditya Parameswaran *Covers Lecture(s): 03–04*

I Relational Algebra

1. **Warmup**: Match each of the following relational algebra operations to the corresponding symbol(s).

a. Defining a relation	I. $R \cap S$
b. Projection	II. $\rho_{S(B_{i_1} \to A_1, B_{i_2} \to A_2, \ldots, B_{i_n} \to A_n)}(R)$
c. Selection	III. $R(B_1, B_2, \ldots, B_m)$
d. Renaming	IV. $\pi_{A_1, A_2, \ldots, A_n}(R)$
e. Union	V. $\rho_{S(A_1, A_2, \ldots, A_n)}(R)$
f. Intersection	VI. $R \cup S$
g. Cross product	VII. $R \bowtie_\theta S$
h. Theta Join	VIII. $R \bowtie S$
i. Natural Join	IX. $R \times S$
	X. $\sigma_C(R)$

2. Consider the following schema for the question below:

   - `books (bid, title, year, genre, aid, times_trending)`
   - `authors (aid, name, publisher, debut_year)`

   Write **relational algebra expressions** for the following queries:

   (a) Find the names of the authors who have books with a genre of either `mystery` or `nonfiction`.

   (b) **[Optional]** Rewrite your answer to part (a) using only the *primitive* relational algebra operators $\{\sigma, \pi, \rho, \times, \cup, -\}$ (i.e., do not use any join operators).

(c) Find the author ids who have books of genre `fantasy` or have books that have trended at least 5 times.

(d) Find the name of the authors who have not released any books. (Note: authors who have not released any books will still have a debut year.)

(e) **[Optional]** Find the names of the authors who have books with a genre of both `mystery` and `nonfiction`.

II Subqueries

In the previous section, the queries we have written just collect a set of rows by combining data from multiple tables. But sometimes, we need to inspect another table *inside* the logic for a query. **Subqueries** let us do this by using queries inside other queries.

3. Subqueries are useful when we want to find data such that there are (or are not) rows in the result of another query based on that data. We can accomplish this using the `WHERE (NOT) IN` construct. Write a query to find the names of all `authors` who have not written any book.

4. Write a query to find the names of books that were the only one in that year. (Hint: This is equivalent to finding all books `x` such that there does not exist another book `y` such that `y.year = x.year`.)

5. [**Optional, Challenge**]: Write a query to find the titles of books that were the only winner of an award in a given year, i.e., no other book received the same award in the same year.

   - `books (bid, title, year, genre, aid, times_trending)`
   - `awards (award_id, bid, award_name, year)`

III Common Table Expressions

Consider the following schema for the questions below:

- `books (bid, title, year, genre, aid, times_trending)`
- `authors (aid, name, publisher, debut_year)`

So far, our queries have been defined as single expressions with no way to reuse relations. In iterative programming languages, we declare variables and functions to create such reusable abstractions. What about in SQL?

If we had proper variables (like in Python), it could look something like:

```
def overall_query(...):
    books = cte_query(...)
    # something with books
overall_query()
```

In SQL, the equivalent syntax, called a **common table expression (CTE)**, is:

```
WITH books AS
  (SELECT ... /* cte_query */)
SELECT ... /* overall_query */
```

Suppose we would like to find the names of authors who debuted after 2018 and wrote books in 2020.

6. [**Optional**] Write out a query **without** using CTEs that computes the names of unique authors who debuted after 2018 and wrote at least a book in 2020.

7. Write out a query that uses a CTE which temporarily defines `relevant_authors` as all authors who debuted after 2018, and then computes the unique genres of books written by authors in `relevant_authors`.

8. Do Question 6 using CTEs to define `relevant_books` (published in 2020) and combine `relevant_authors` (debuted after 2018) from Question 7. What are the pros and cons of using CTEs in this case?

IV Joins [Optional]

Suppose the `books` and `authors` tables are populated with the following tuples:

author_id	author_name
1	Agatha Christie
2	bell hooks
3	Chanwoo
4	Desmond Tutu

`authors`

book_id	book_name	author_id
1	And Then There Were None	1
2	Endless Night	1
3	Teaching to Transgress	2
4	Chanwoo's Photobook	3
5	Nothing Personal	5

`books`

9. **Warmup**: Matching each join operator below with the appropriate table result.

   (a) Inner join _____

   (b) Left join _____

   (c) Outer join _____

Table results:

I.

author_id	author_name	book_id	book_name	author_id
1	Agatha Christie	1	And Then There Were None	1
1	Agatha Christie	2	Endless Night	1
2	bell hooks	3	Teaching to Transgress	2
3	Chanwoo	4	Chanwoo's Photobook	3
4	Desmond Tutu	NULL	NULL	NULL

II.

author_id	author_name	book_id	book_name	author_id
1	Agatha Christie	1	And Then There Were None	1
1	Agatha Christie	2	Endless Night	1
2	bell hooks	3	Teaching to Transgress	2
3	Chanwoo	4	Chanwoo's Photobook	3
NULL	NULL	5	Nothing Personal	5

III.

author_id	author_name	book_id	book_name	author_id
1	Agatha Christie	1	And Then There Were None	1
1	Agatha Christie	2	Endless Night	1
2	bell hooks	3	Teaching to Transgress	2
3	Chanwoo	4	Chanwoo's Photobook	3

IV.

author_id	author_name	book_id	book_name	author_id
1	Agatha Christie	1	And Then There Were None	1
1	Agatha Christie	2	Endless Night	1
2	bell hooks	3	Teaching to Transgress	2
3	Chanwoo	4	Chanwoo's Photobook	3
4	Desmond Tutu	NULL	NULL	NULL
NULL	NULL	5	Nothing Personal	5

10. **[Optional]** Of the joins in the previous question, which can be written as a Select-From-Where (SFW) query *without* using a JOIN keyword? Hint: What happens when we specify multiple source tables in the FROM clause?

V Set and Bag Operations [Optional]

Query execution has its foundation in Relational Algebra, which shares some of its concepts with Set Theory. *Sets* are unordered collections of items that do not contain duplicates, i.e., multiple instances of the same element. On the other hand, *Bags* are like sets, but allow for duplicates. You may already have an idea of how this concept relates to query processing! (Hint: duplicate rows)
Let's take some time to familiarize ourselves with set notation, particularly if it is new to you:

- Let A and B be *sets* where $A = \{1, 2, 3, 4, 5\}$ and $B = \{2, 3, 4, 5, 6\}$.
- Let R and S be *bags* where $R = \{1, 1, 1, 2, 2, 4, 4\}$, and $S = \{2, 3, 3, 3, 4, 4\}$.

11. *Union.* Write out $A \cup B$ and $R \cup S$.

12. *Intersection.* Write out $A \cap B$ and $R \cap S$.

13. *Difference.* Write out $A - B$ and $R - S$.

14. *Difference.* Write out $B - A$ and $S - R$.

15. *Commutativity.* Based on your analysis above, are the union, intersection, and difference operator commutative on sets and bags? Recall that *commutativity* means changing the order of the operands does not change the result, e.g., in algebra, $x + y = y + x$.

16. Suppose we have SQL relations S and R with the same schema. How would we write a SQL query that computes $S \cup R$ as a *bag*?

17. Repeat the previous question, but with (a) $S \cup R$ as a set, (b) $S - R$ as a set, and (c) $S - R$ as a bag.

VI Virtual & Materialized Views [Optional]

CTEs are great, but they can only be used in the context of a single query. If we have an intermediate result we want to use across separately issued queries, we need the equivalent of *functions*. In SQL, we can create these as **virtual views** and **materialized views**. Syntactically, creating a SQL view (materialized or otherwise) is very similar to creating a new SQL table from an existing table. However, depending on your use case—frequency of access, frequency of updates, etc.—you may choose one over the other.

18. Suppose a national bookstore database maintains two tables:

    - `books(title, author, genre, isbn, pub_date)` has several million records, one per book. It is updated daily (`pub_date` is a calendar date) to include each newly published book available to order, and it is indexed on `isbn`.
    - `purchases(isbn, count, total_revenue)` has one record per book in `books`. It is updated every time a customer makes a purchase in any store, possibly several times a second, and it is indexed on `isbn`.

    For each of the below scenarios, would you use `books` and `purchases` to (I) create a view, (II) create a materialized view, (III) create a new table, or (IV) write a regular SELECT query? Why?

    (a) Find the books published in the last calendar month (e.g., if today is September 12, 2024, then find books published August 1–31, 2024). This result is accessed every second.

    (b) Find the 5 top-selling books. This result is accessed every second.

    (c) Find all books by a specific author. This result is accessed every second (for different authors). *Note:* This is a more general scenario than the previous parts; assume there are several hundred thousand possible authors that can be specified.

    (d) Find a specific author by name in the database. This result is accessed every second (for different authors).