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: $\underline{R(B_1, B_2, \ldots, B_m)}$
   b. Projection: $\underline{\pi_{A_1,A_2,\ldots,A_n}(R)}$
   c. Selection: $\underline{\sigma_C(R)}$
   d. Renaming: $\underline{\rho_{S(A_1,A_2,\ldots,A_n)}(R)}$ or $\underline{\rho_{S(B_{i_1}\to A_1,B_{i_2}\to A_2,\ldots,B_{i_n}\to A_n)}(R)}$
   e. Union: $\underline{R \cup S}$
   f. Intersection: $\underline{R \cap S}$
   g. Cross product: $\underline{R \times S}$
   h. Theta Join: $\underline{R \bowtie_\theta S}$
   i. Natural Join: $\underline{R \bowtie S}$

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`.

   $\pi_{authors.name}(authors \bowtie_{authors.aid=books.aid} \sigma_{genre=\text{`mystery'} \vee genre=\text{`nonfiction'}}(books))$

   We want to find mystery or nonfiction books so we use the selection operator with the appropriate selection condition on the books relation. Since we want to find the names of the authors who wrote those books, we do a theta-join on the authors relation with the result of the earlier selection operator. Finally, we project away all the columns except for the author name column with the projection operator.

   (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).

   $\pi_{\text{authors.name}}\Big(\sigma_{\text{authors.aid=books.aid} \wedge (\text{genre=`mystery'} \vee \text{genre=`nonfiction'})}(\text{ authors} \times \text{books })\Big)$

   Since we are restricted to primitives, we replace the join with a selection over a Cartesian product: a θ-join can be written as $\sigma_\theta(\text{authors} \times \text{books})$. We take the product of `authors` and

`books`, then use a selection to enforce the key match `authors.aid = books.aid` and keep only rows where the book's genre is mystery or nonfiction. Finally, we project `authors.name` to keep just the author names (projection removes duplicates under set semantics).

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

$\pi_{books.aid}(\sigma_{genre='fantasy'}(books)) \cup \pi_{books.aid}(\sigma_{times\_trending>=5}(books))$

We want fantasy genre books so we use a selection operator with the appropriate selection condition on the books relation, and then project away all columns except for the author id column. We also do a selection on books that have trended at least 5 times, and then project away all columns except for the author id column. We finally take the union of the result of the two projections to get our final answer.

(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.)

$\pi_{authors.name}(authors \bowtie (\pi_{aid}(authors) - \pi_{aid}(books)))$

Note here that all records in the books relation are authors who have released books. On the other hand, records in the authors relation include authors who have released books and also authors who have not released books.

With this in mind, we know we want to find the difference between two relations. So we first project away all unnecessary columns, keeping the author id column for the authors relation and the books relation. We then take the difference between the result of the two projections, and then join it with the authors relation in order to take a projection of the author name column.

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

$\pi_{authors.name}(authors \bowtie_{authors.aid=books.aid} \sigma_{books.genre='\texttt{mystery}'}(books)) \cap$
$\pi_{authors.name}(authors \bowtie_{authors.aid=books.aid} \sigma_{books.genre='\texttt{nonfiction}'}(books))$

We want mystery books so we use the selection operator for the mystery genre. We then join the selection result with the authors relation and project away all columns except for the author name column. We repeat the same process for the nonfiction genre books, and join the result of the two projections with an intersection operator because we want to find authors who have released books in both genre categories.

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.

```
SELECT name
FROM authors
WHERE aid NOT IN (SELECT aid FROM books);
```

We first use a subquery to find all author IDs who appear in the `books` table; those are the authors who have authored books. Then, our main query retrieves the names of the authors in `authors` who are not in this set.

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`.)

```
SELECT b1.title
FROM books AS b1
WHERE NOT EXISTS (SELECT * FROM books AS b2 WHERE b1.bid != b2.bid AND b1.year =
b2.year);
```

If a book is the only one in its year, that means that no other book (with a different ID) exists in that 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)`

Solution 1:

```
SELECT b.title
FROM books AS b
INNER JOIN awards AS a ON b.bid = a.bid
WHERE NOT EXISTS (
SELECT *
FROM books AS b2
INNER JOIN awards AS a2 ON b2.bid = a2.bid
WHERE b.bid != b2.bid AND a.year = a2.year AND a.award_name = a2.award_name
);
```

Solution 2:

```
SELECT b.title
FROM books AS b
```

```
INNER JOIN awards AS a ON b.bid = a.bid
WHERE NOT EXISTS (
SELECT *
FROM awards AS a2
WHERE b.bid != a2.bid
AND a.year = a2.year AND a.award_name = a2.award_name
);
```

In Solution 1, we use a subquery that joins the books and awards tables again to check if there exists any other book `b2` that won the same award `a2.award_name` in the same year `a2.year`. If such a book is found, the query excludes the current book `b` from the results.

In Solution 2, we simplify the subquery by only querying the awards table. Here, we check if another book `a2.bid` from the same year and with the same award exists, without rejoining with the books table. This approach is more efficient, as it avoids an unnecessary join and directly compares the relevant information within the awards table itself.

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.

```
SELECT DISTINCT a.name
FROM authors AS a
INNER JOIN books AS b
ON a.aid = b.aid
WHERE a.debut_year > 2018 AND b.year = 2020;
```

We first join `authors` with their `books` and look for their books. The base query could actually return duplicate names if an author has written multiple books in 2020, so we use the DISTINCT keyword to return a set.

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`.

```
WITH relevant_authors AS (
SELECT *
FROM authors
WHERE debut_year > 2018)

SELECT DISTINCT b.genre
FROM relevant_authors AS a
INNER JOIN books AS b
ON a.aid = b.aid;
```

Here, we break up the query to filter `authors` by debut year first before joining with `books`.

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?

```
WITH relevant_authors AS (
SELECT *
FROM authors
WHERE debut_year > 2018),

relevant_books AS (
SELECT *
FROM books
WHERE year = 2020)

SELECT DISTINCT a.name
FROM relevant_authors AS a
INNER JOIN relevant_books AS b
ON a.aid = b.aid;
```

Effectively, all we've done is push the filtering clauses into the CTEs, so that we have a simple theta join at the end.

- Pros: improves the readability of the query
- Cons: more verbose since we have to write the entire query for each CTE

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

a. Inner join: (III) Notice that the inner join result never contains `NULL` on the join predicate columns. In this example, Desmond Tutu does not appear in the result because he has not authored a book in `books` (Desmond Tutu has written several books, including co-writing *The Book of Joy* with the Dalai Lama). *Nothing Personal* also does not appear because its author (James Baldwin) is not found in the authors table.

b. Left join: (I) Notice that all tuples from the left table are present in the result, even though some values are null because they have no matching rows in the right table.

c. Outer join: (IV) This table represents the result of the full outer join query between the authors and books tables. It includes all rows from both tables, matching rows based on the `author_id` values. If there is no match, the corresponding columns are filled with null values to indicate the absence of data.

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?

Inner join, because it can be written with a cross product, and returns only rows that have valid tuples in both source tables (as opposed to "NULL" tuples). Here's how it works:

- **Multiple tables in FROM clause:** this creates a Cartesian product of these tables, which means that SQL combines every row from the first table with every row from the second table, resulting in a large number of possible combinations.
- **Matching rows with WHERE clause:** the condition in the WHERE clause filters this combination to include only rows where values match, which is the same as an inner join.

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 $R \cup S$.

> $A \cup B = \{1, 2, 3, 4, 5, 6\}$.
> $R \cup S = \{1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4\}$.
>
> The union of two sets/bags contains all elements that are members of either set/bag.

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

> $A \cap B = \{2, 3, 4, 5\}$.
> The intersection of two sets contains all *unique* elements that are members of both sets.
> $R \cap S = \{2, 4, 4\}$.
> The intersection of two bags contains the elements that are members of both bags. The *multiplicity* of each element (i.e., the number of times it appears) in the resulting bag is its minimum multiplicity across both bags.

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

> $A - B = \{1\}$.
> The difference of two sets contains all unique elements in the first set that are not in the second set.
> $R - S = \{1, 1, 1, 2\}$.
> The difference of two bags contains all elements in the first bag that are not in the second bag. The multiplicity of each element is given by the difference of its multiplicities in each bag.

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

> $B - A = \{6\}$.
> $S - R = \{3, 3, 3\}$.

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, $x + y = y + x$.

> For both sets and bags, the union and intersection operators are commutative. However, (just like in algebra,) the difference operator is not.

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*?

$S \cup R$ as a bag:

```
SELECT * FROM S
UNION ALL
SELECT * FROM R;
```

`UNION ALL` performs a union of the two relations but retains all duplicates.

Note this implies that the default function of `UNION` is a setwise union, i.e., `UNION` and `UNION DISTINCT` are equivalent.

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.

- $S \cup R$ as a set:

  ```
  SELECT * FROM S
  UNION
  SELECT * FROM R;
  ```

  UNION performs a union of the two relations and eliminates duplicate rows from its result.

- $S - R$ as a bag:

  ```
  SELECT * FROM S
  EXCEPT ALL
  SELECT * FROM R;
  ```

  EXCEPT ALL returns all rows that are in relation S but not in relation R, and retains duplicate rows.

- $S - R$ as a set:

  ```
  SELECT * FROM S
  EXCEPT
  SELECT * FROM R;
  ```

  EXCEPT returns all rows that are in relation S but not in relation R, and eliminates duplicate rows.

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).

A range of answers are possible after some discussion, but here are some possible choices and reasonable justification:

(a) Materialized view, if supported. The result is accessed much more frequently than the table is being updated, so we can avoid some recomputation using materialization.

(b) View. The result is accessed about as frequently as the table updates, so we cannot avoid any recomputation if we want everything to be up-to-date. (Note: realistically, if this were customer-facing, we may opt for a more approximate "top-seller" list, e.g., updated daily or weekly, suggesting materialization if possible).

(c) Query. It would be unrealistic to create a view for every possible author that could be queried. Instead, it is more efficient and less work to provide a regular SELECT query.

(d) New table of authors. The intended scenario asks about authors (as opposed to books), so semantically (if we are considering the larger database design) having a separate table storage for authors would be reasonable. Note that by creating this new table, a new book by a new author would necessarily generate updates to both `books` and `authors`.