A Step-by-Step Guide to Writing Queries in Relational Algebra

Prof. Fedeli Massimo – IIS Fermi Sacconi Cpia

Introduction

Relational Algebra is a procedural query language that forms the mathematical foundation of relational database systems. Introduced by Edgar F. Codd in his seminal 1970 paper "A Relational Model of Data for Large Shared Data Banks," it provides a formal framework for manipulating and retrieving data stored in tables (relations). Unlike SQL—which is declarative and focuses on *what* data to retrieve—relational algebra is *procedural*: it describes *how* to obtain the desired result through a sequence of well-defined operations.

The core operators of relational algebra include:

- **Selection** (σ): filters rows based on a condition.
- **Projection** (π): selects specific columns.
- **Union** ($\cup$), **Difference** ($-$), and **Intersection** ($\cap$): combine or compare sets of tuples.
- **Cartesian Product** ($\times$) and **Join** ($\bowtie$): combine data from multiple relations.
- **Rename** (ρ): changes relation or attribute names.
- (Extended) **Aggregation** (γ) and **Division** ($\div$): support grouping and universal quantification.

Why learn relational algebra today?

- **Conceptual clarity**: It helps you understand *how* databases work under the hood, beyond writing SQL statements.
- **Query optimization**: Database management systems (DBMS) internally translate SQL queries into relational algebra expressions to optimize execution plans.
- **Foundation for advanced topics**: Knowledge of relational algebra is essential for studying database theory, query languages, and data integrity.
- **Problem-solving skill**: Translating natural-language questions into formal expressions sharpens logical and analytical thinking—skills valuable in programming, data science, and software engineering.

This guide is designed for students encountering relational algebra for the first time. It assumes no prior knowledge beyond basic familiarity with tables and relationships. Using the **UIBK Music Streaming Service** database—a realistic, pedagogical schema inspired by real-world applications—we will walk through a systematic, step-by-step method to construct correct and efficient relational algebra queries.

By mastering this formalism, you will not only become better at writing SQL but also develop a deeper understanding of how data is structured, connected, and retrieved in modern information systems.

The Mental Algorithm: How to Build a Relational Algebra Query

Before diving into details, keep this 5-step mental algorithm in mind. Think of it as your "recipe" for every query:

Step 1: What do I need to output? Identify the attributes (columns) the query asks for (e.g., names, IDs). This tells you what to put in the final *projection* (π).

Step 2: Where is this data stored? List all relations (tables) that contain the required attributes or are needed to connect them (via foreign keys).

Step 3: What conditions must be satisfied? Identify filters (e.g., "from Italy", "genre = Rock"). These become *selections* (σ).

Step 4: How do I connect the tables? Use *joins* ($\bowtie$) to combine relations based on matching keys (e.g., `CustomerId`, `TrackId`).

Step 5: Do I need advanced logic? Ask: Is this a "not", "all", or "count/average" query? If yes, consider *difference* ($-$), *division* ($\div$), or *aggregation* (γ).

Once you've answered these five questions, you can assemble your expression **from the inside out**: start with selections, then join, then project.

Step 1: Understand the Question

Before writing any symbols, read the query carefully. Ask yourself:

- What information am I asked to return? (e.g., names, IDs, counts?)
- Which tables (relations) contain this information?
- Are there conditions or filters? (e.g., "from Italy", "price ¿ 10")
- Do I need to combine data from multiple tables?

Example: "Find the name of all customers who live in Italy."

- Output: `FirstName` and `LastName`
- Table: `Customer`
- Condition: `Country = 'Italy'`

Step 2: Reference Schema — UIBK Music Streaming Service Database

All queries in this guide refer to the following relational schema:

- **Artist**(*ArtistId*, *Name*)
- **Album**(*AlbumId*, *Title*, *ArtistId*)
- **Track**(*TrackId*, *Name*, *AlbumId*, *GenreId*, *Milliseconds*, *Bytes*, *UnitPrice*)
- **PlaylistContent**(*PlaylistId*, *TrackId*)
- **Playlist**(*PlaylistId*, *Name*)
- **Genre**(*GenreId*, *Name*)
- **Invoice**(*InvoiceId*, *CustomerId*, *InvoiceDate*, *Total*)
- **InvoiceParts**(*InvoicePartId*, *InvoiceId*, *TrackId*, *UnitPrice*, *Quantity*)
- **Customer**(*CustomerId*, *FirstName*, *LastName*, *Address*, *City*, *Country*, *PostalCode*, *Email*)

Key Relationships:

- `Album.ArtistId` → `Artist.ArtistId`
- `Track.AlbumId` → `Album.AlbumId`
- `Track.GenreId` → `Genre.GenreId`
- `PlaylistContent.PlaylistId` → `Playlist.PlaylistId`
- `PlaylistContent.TrackId` → `Track.TrackId`
- `Invoice.CustomerId` → `Customer.CustomerId`
- `InvoiceParts.InvoiceId` → `Invoice.InvoiceId`
- `InvoiceParts.TrackId` → `Track.TrackId`

Tip: Always check foreign key relationships when joining tables!

Step 3: Apply Selection (σ) to Filter Rows

Use the **selection operator** $\sigma_{\text{condition}}(R)$ to keep only the rows that satisfy a condition.

Example: Customers from Italy:

$$\sigma_{\text{Country='Italy'}}(\text{Customer})$$

Step 4: Combine Relations with Join ($\bowtie$)

When data is split across tables, you must combine them to answer complex questions. In relational algebra, the most common way to do this is the **natural join** ($\bowtie$), which automatically matches tuples with equal values on attributes that share the same name (e.g., `CustomerId`).

However, in practice—and especially when attribute names differ—you may need to specify the join condition explicitly. While pure relational algebra uses natural join, many textbooks and courses adopt a **theta-join** notation:

$$R \bowtie_\theta S$$

where θ is a condition (e.g., $R.A = S.B$).

Below, we explain the main types of joins you should know.

Types of Joins in Relational Algebra

Although classical relational algebra defines only the **natural join**, it is useful to understand how it relates to other join types commonly used in SQL and database theory.

Natural Join ($R \bowtie S$) Combines tuples from R and S that have **equal values on all attributes with the same name**. The common attributes appear only once in the result. **Example**: Customer $\bowtie$ Invoice joins on `CustomerId` (present in both relations) and returns one `CustomerId` column.

Theta-Join ($R \bowtie_\theta S$) A generalization where you specify a condition θ (e.g., equality, inequality). In practice, this is how we join tables when keys have different names or when using non-equality conditions. **Example**: Track $\bowtie_{\text{Track.GenreId=Genre.GenreId}}$ Genre

Equijoin A theta-join where θ consists only of equality conditions on attributes. Most foreign-key joins are equijoins. Natural join is a special case of equijoin where duplicate columns are removed.

Outer Joins (Left, Right, Full) *Note*: These are **not part of classical relational algebra**, which operates only on complete tuples (no nulls). However, they are essential in SQL. - **Left outer join**: keeps all tuples from the left relation, filling missing right-side data with NULL. - Used when you want "all artists, even those without albums." Since relational algebra does not support NULLs, such queries are often expressed using **union** and **difference** instead. **Example (simulated)**: To find all artists and their albums (including artists with no albums):

$$\pi_{\text{Name, Title}}((\text{Artist} \bowtie \text{Album})) \cup \pi_{\text{Name, Title}\leftarrow\text{NULL}}(\text{Artist} - \pi_{\text{ArtistId}}(\text{Album}))$$

(This is advanced; usually, standard curriculum focuses on inner/natural joins.)

Practical Advice for This Course: In exercises based on the UIBK schema, assume that:

- All joins are **inner joins** (only matching tuples are kept).
- When two tables share a foreign key (e.g., `Track.GenreId` and `Genre.GenreId`), use an explicit equijoin if names differ, or natural join if you rename attributes first (using ρ).
- For simplicity, we often write: Track $\bowtie$ Genre implying the join on `GenreId`, even if attribute names are technically different. In formal settings, use renaming or theta-join.

Step 5: Project Only the Required Attributes (π)

Use the **projection operator** $\pi_{\text{attr}_1,\ldots,\text{attr}_n}(R)$ to keep only the columns you need in the final result.

Example: Return only track names:

$$\pi_{\text{Name}}\Big((((\sigma_{\text{Country}='Italy'}(\text{Customer})) \bowtie \text{Invoice}) \bowtie \text{InvoiceParts}) \bowtie \text{Track}\Big)$$

Step 6: Handle Advanced Cases (Optional)

Some queries require more advanced operators:

Difference ($-$): Find items in one set but not another. Example: "Tracks never purchased" = All tracks $-$ Purchased tracks.

Division ($\div$): Used for "for all" queries. Example: "Customers who bought tracks from every genre".

Aggregation (γ): Compute sums, averages, counts. Example: $\gamma_{\text{avg(UnitPrice)}}(\text{Track})$

Step 7: Simplify and Check

- Can you reduce the number of joins?
- Are all attributes correctly named?
- Does the output match what the question asks?

Tip: Work backwards—start from the desired output, then ask: "Where does this data come from?"

Worked Example

Query: "Find the name of all playlists that contain at least one track from the 'Rock' genre."

1. Needed output: `Playlist.Name`
2. Tables involved: `Playlist`, `PlaylistContent`, `Track`, `Genre`
3. Conditions: `Genre.Name = 'Rock'`
4. Steps:
   - Select Rock genre: $\sigma_{\text{Name}='Rock'}(\text{Genre})$
   - Join with Track: Track $\bowtie_{\text{Track.GenreId = Genre.GenreId}} (\cdots)$
   - Join with PlaylistContent: on `TrackId`
   - Join with Playlist: on `PlaylistId`
   - Project `Name` from Playlist
5. Final expression:

$$\pi_{\text{Playlist.Name}}\Big(\text{Playlist} \bowtie \text{PlaylistContent} \bowtie \text{Track} \bowtie \sigma_{\text{Name}='Rock'}(\text{Genre})\Big)$$

Summary Checklist

- ☐ Understood the natural-language query?
- ☐ Identified all necessary relations? (Refer to Section 2!)
- ☐ Applied selections (σ) to filter rows?
- ☐ Joined relations correctly using foreign keys?
- ☐ Projected only the required attributes (π)?
- ☐ Considered advanced operators if needed (e.g., $-$, $\div$, γ)?

With practice, writing relational algebra expressions will become intuitive—and it will greatly improve your ability to write efficient SQL queries!

References

1. R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed. New York: McGraw-Hill, 2003. (Chapter 4: Relational Algebra and Calculus)
2. A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York: McGraw-Hill, 2019. (Section 6.1–6.2)
3. C. J. Date, *An Introduction to Database Systems*, 8th ed. Boston: Addison-Wesley, 2003.
4. University of Innsbruck (UIBK), Department of Databases and Information Systems. Course materials on Relational Algebra. Available: https://www.dbs.uibk.ac.at/
5. “Chinook Database.” GitHub Repository, 2023. [Online]. Available: https://github.com/lerocha/chinook-database. (Note: The UIBK Music Streaming schema used in this guide is inspired by the Chinook sample database.)
6. Stanford University, “Relational Algebra,” CS145: Introduction to Databases. [Online]. Available: https://cs145-fa22.github.io/slides/04-relational-algebra.pdf