Data I/O and Preprocessing with SQL and Python

Module 4: Preprocessing, validation, and joins with SQL

DeepLearning.AI

Preprocessing, validation, and joins with SQL

Module 4 introduction

DeepLearning.AI

Module 4 outline

Filtering and conditionals in SQL

- Select subsets using `WHERE`
- Conditionals: `IN` and `BETWEEN`
- String-based conditionals

Data validation

- Resolving missing & unexpected data
- Grouping
- Aggregation functions: `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`
- Filtering **after** grouping

Types of joins

- Left joins
- Inner joins
- Outer joins

Preprocessing, validation, and joins with SQL

SQL vs Python

DeepLearning.AI

When to use SQL

- **Generally:**
    - Use SQL when it is most efficient
    - Use Python when you need a more custom analysis
    - In practice:
        - 📈 Often be working with massive datasets of millions or billions of rows
        - ❌ Pulling all rows across network wastes time
- **Using SQL:**
    - Select only necessary rows for analysis
    - Grouping, joins, and sorting
    - Working in collaborative contexts
        - Maintain standardized set of queries
        - Keep interpretability consistent
    - Queries can also be scheduled
        - **Example:** Fetch data about all shipping orders in the past 7 days
        - Easier to maintain compared to Python notebook

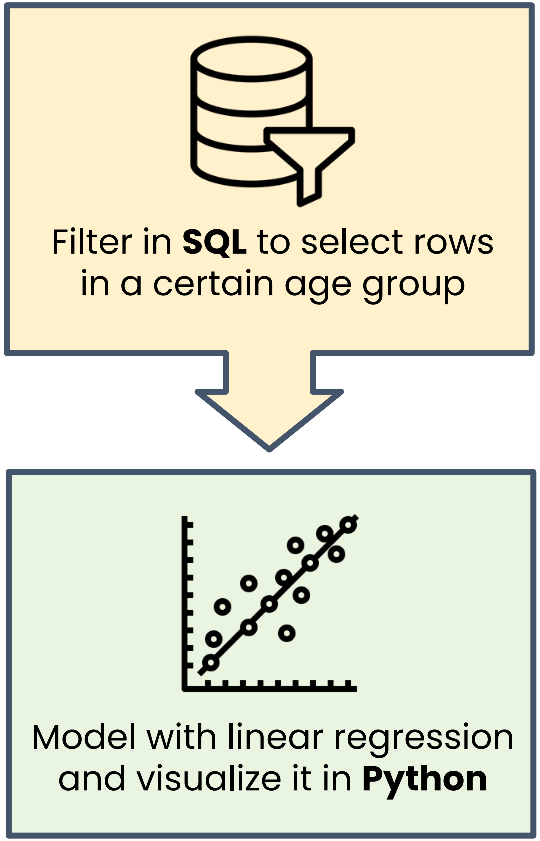
When to use Python

- ✅ More advanced analysis
- ✅ Efficiency and repeatability are less of a priority
- ✅ Advanced and custom analyses
  - Box plot
  - Train a linear regression model
- ✅ If you have flat files or working with data from an API

- **Example**: Exploration mode
  - 📊 Quick visualizations
  - ⚡ Many analyses quickly
  - 👎 Can feel clunky in SQL

Hybrid workflow

- **Example**: Report about weekly active users by age group on messaging platform
- Python and SQL are **complementary,** not mutually exclusive!



Preprocessing, validation, and joins with SQL

Filtering

DeepLearning.AI

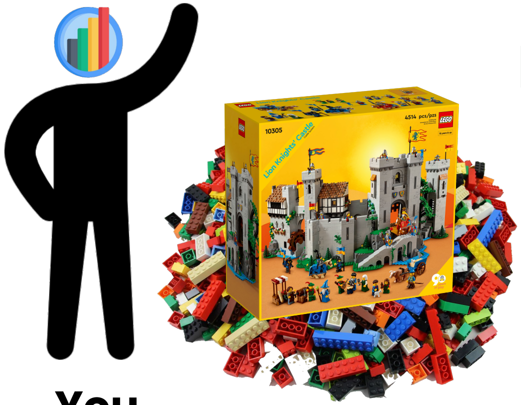
Filtering

- SQL filtering works very similarly to filtering in Python
- You'll use:

```
WHERE condition;
```

  - Query will only return rows that meet that condition

Scenario



You
Data Analyst

🏆 **Goal:** Identify trends in LEGO sets, like themes and sizes

📊 **Data:** SQLite database of LEGO sets released 1950-2017

- Stored in file `lego_sets.db`
- Working with the table `sets_with_themes`

🎯 **Task**: Identify popular LEGO set themes over the years

- Interested in: Largest sets, which tend to be more expensive

Lion Knights' Castle 10305, 2025. *Lion Knights' Castle*. *@LEGO*

Recap: Filtering

- You learned two different methods for filtering data in SQL
- Both filtering methods use:

```
WHERE conditional ;
```

  - **WHERE** clause
  - With different conditional

Method 1: To filter for values less than, greater than, or equal to a particular value

```
SELECT *
FROM sets_with_themes
WHERE num_parts > 1000;
```

Method 2: To filter for multiple specific values

```
SELECT *
FROM sets_with_themes
WHERE year IN (1999, 2001);
```

Preprocessing, validation, and joins with SQL

Filtering: Compound conditions

DeepLearning.AI

Compound conditionals

- Filter rows based on complex conditions using logical operators:
  - `AND` - returns rows where **all** conditions are true
  - `OR` - returns rows where **at least one** condition is true
- These operators allow you to filter rows by multiple columns

Scenario:

📊 **Data:** LEGO set database

🎯 **Task**: Filter for sets according to both theme and year

- Specific focus: Aquatic themes

Recap: Compound conditionals

To write queries with compound conditions:

```
SELECT *
FROM sets_with_themes
WHERE (theme_name = "Aquanauts" OR theme_name = "Aquasharks") AND num_parts > 100
ORDER BY theme_name;
```

- SQL evaluates `AND` before `OR`:
  - `AND` - stricter, more specific
  - `OR` - allows more flexibility
- To avoid silent logical errors, use parentheses to group conditionals

It's sunny **AND** Before lunch

Both conditions must be true

If it's sunny **OR** Before lunch

Either condition is enough

Preprocessing, validation, and joins with SQL

Filtering: String-based conditions

DeepLearning.AI

LIKE operator

- Flexible way to filter text data using patterns
- To exactly match value:

```
WHERE name LIKE "castle";
```

- Two special character called wildcards:
  - `%` - matches any sequence of **one or more characters**
  - `_` - matches **exactly one character**
- You can’t define patterns with same level of specificity as regular expressions

Scenario:

📊 **Data:** LEGO set database

🎯 **Task**: Finding all sets with castle in their name

- Sets have the word “castle” in different locations in their names

Recap: String-based conditions

- To match patterns in strings:

```
SELECT *
FROM sets_with_themes
WHERE name LIKE "castle%";
```

  - **Example**: Search all sets whose names start with castle

⛔ Wildcards can slow down queries

⛔ **%** at beginning of pattern can create inefficiencies

⛔ Avoid leading wildcards if you can

- If you frequently search same patterns:
  - Separate column with tags or categories that you can filter directly on
- **Example**:
  - Frequently searching for sets with minifigures (i.e. LEGO people)
  - Solution: Add new column in dataset with that information

Preprocessing, validation, and joins with SQL

Conditionals: CASE

DeepLearning.AI

SQL conditionals

- Function similarly to Python if and else statements
- Add decision-making logic directly into queries
- You can:
  - ⬜ Evaluate conditions
  - ⬜ Return specific results based on those conditions

Scenario:

📊 **Data:** LEGO set database

🎯 **Task**: Categorize sets based on size

- Small
- Medium
- Large

🏆 **Goal:** Better organize them

Recap: CASE

- Write conditionals using a CASE statement:

```
SELECT name, num_parts,
CASE
  WHEN num_parts < 10 THEN "Small"
  WHEN num_parts BETWEEN 10 AND 172 THEN "Medium"
  ELSE "Large"
END AS set_size
FROM sets_with_themes;
```

1. Creates a new column with values based on conditions
2. WHEN a certain condition is true, THEN add specified value to column.
3. END case statement and give column a name using AS.

- Conditionals are great for:
  - ✅ Efficiently segmenting data
  - ✅ Reducing number of values in analysis or visualization

Preprocessing, validation, and joins with SQL

Handling NULL values

Handling NULL values

- **Recall**:
  - 🔍 Nulls are missing values in your code
  - 📗 You've encountered them before in Python
- SQL can represent null values
  - Using keyword `NULL`
- Common to handle in different ways:
  - ⬜ Drop rows that have nulls by filtering
  - ⬜ Fill null values with another value

Recap: Handling NULL values

Select rows with null values in a column:

```
SELECT *
FROM sets_with_themes
WHERE parent_theme_id IS NULL;
```

Select rows that aren't null:

```
SELECT *
FROM sets_with_themes
WHERE parent_theme_id IS NOT NULL;
```

To fill a column with non-null values

```
SELECT theme_id, parent_theme_id,
       COALESCE(parent_theme_id, theme_id) AS updated_parent_theme_id
FROM sets_with_themes;
```

- **Example**: Fill the **`parent_theme_id`** column with the **`theme_id`**, if the **`parent_theme_id`** column was null.

Preprocessing, validation, and joins with SQL

Data validation

DeepLearning.AI

Reasons for validation

Example: Ran a query to retrieve sales data for the month of January

😄 **Expected**: Records for hundreds of transactions

😅 **Actual**: Just a few values

⛔ Wasted significant time on incomplete data

✅ Verify data at each step to ensure it aligns with expectations and goals

Customer ID	Order Date	Sales Amount
1001	2025-01-19	25.00
1001	"January 19, 2025"	200.00
	2025-01-21	-30.00

1️⃣ Unexpected date formats

- Impossible to sort or filter chronologically

2️⃣ Duplicate or missing entries

- Inflates total number of customers
- Leads to incomplete analysis

3️⃣ Unexpected values

- Sales amounts might not match expectations

Think through expectations

📊 **Dataset:**

- Diagnostic measurements for predicting whether a patient has diabetes
- Collected from females of Pima Indian heritage, aged 21 years or older

🤔 Before running query, think through the results you expect:

You should know:

- 📝 1 (diabetes) or 0 (no diabetes)
- 🎂 Only women aged 21 or older
- 🫀 Normal values are for glucose, blood pressure, insulin, and BMI

❌ Other numbers would indicate that something is wrong with data

❌ Other age values may be mistakes, or initial understanding was incorrect

❌ Values that would be impossible have: sign that there's a problem

Validate with subsets

- When working with massive dataset:
  - Unnecessary to load and analyze entire dataset right away
- By testing small samples first, you can:
  - Validate the structure of the data
  - Identify missing values or inconsistencies
  - Test logic to ensure it works

```sql
SELECT *
FROM diabetes
LIMIT 10;
```

```sql
SELECT *
FROM diabetes
WHERE age > 60 AND glucose < 100
LIMIT 10;
```

If results don't match expectations

1. Begin with reviewing your query logic

```
SELECT BMI, Age, Outcome
FROM diabetes
WHERE age > 60
LIMIT 10;
```

- Double-check that the condition is correctly applied

2. Use your dataset's metadata to review what values you should expect
   - Data engineers, or members of data team with more knowledge may be able to help

3. Test alternative queries or use visualizations to better understand discrepancies
   - Unexpected values:
     ✅ Aren't always wrong
     ✅ Might reveal something important about your data

Preprocessing, validation, and joins with SQL

Validation: COUNT, DISTINCT

DeepLearning.AI

COUNT and DISTINCT

COUNT

- “How many records do I have?”
- Comparing to expectations can reveal:
  - Whether data is missing
  - If duplicate rows might exist

DISTINCT

- Identifies unique values in a column
- Helps you understand variety in data
- Useful for detecting duplicate entries

Validation checklist

- ⬜ Start with an intuition about what data should look like
- ⬜ Confirm and refine your understanding:
    - ⬜ **`COUNT`** - to confirm total number of rows or number of non-null values in column
    - ⬜ **`DISTINCT`** - to explore uniqueness and variety in data
    - ⬜ **`COUNT`** and **`DISTINCT`** - validate key characteristics (e.g. completeness)

Preprocessing, validation, and joins with SQL

Validation: GROUP BY

DeepLearning.AI

Scenario

You
Data Analyst

🎯 **Task**: Analyze how lego sets are distributed among different themes

- ⬜ Group data by themes
- ⬜ Analyze characteristics of those groups

✅ Accomplish task using `GROUP BY` statement

Recap: GROUP BY

- Group rows using the **`GROUP BY`** statement and name of a column

```
SELECT theme_name, COUNT(DISTINCT theme_id) AS theme_count
FROM sets_with_themes
GROUP BY theme_name
```

- **Aggregation functions** will apply to each group individually
- Not required to select the column you group by

Order of operations

1. `FROM` – identifying the table the data should come from
2. `WHERE` – filtering
3. `GROUP BY` – grouping
4. Aggregations (e.g. `COUNT`())
5. `SELECT` – grabbing only columns or aggregations you asked for
6. `ORDER BY` – sorting

Preprocessing, validation, and joins with SQL

Validation: MIN, MAX, SUM

DeepLearning.AI

Aggregation functions

- To analyze grouped data
- Work in tandem with `GROUP BY` to organize dataset into groups
- Python's `groupby()`:
    - Doesn't display much
    - Need summarization functions, like `sum()` or `mean()`
- SQL `GROUP BY` combined with aggregations generates summary-level calculations

Key aggregate functions

- **`COUNT`** - counts number of rows that match criteria
- **`AVG`** - average value of column for each group
- **`SUM`** - totals values in a column for each group
- **`MIN/MAX`** - find smallest and largest values within each group

Recap: MIN, MAX, SUM

- SQL offers many aggregation functions, including `MIN()`, `MAX()`, `AVG()`
- By selecting all columns, you can learn more about row containing:
  - Minimum value in column:

    ```
    SELECT *, MIN(num_parts)
    ```

  - Maximum value in column:

    ```
    SELECT *, MAX(num_parts)
    ```

- Many aggregation functions can be used together in same query

```
SELECT theme_name, COUNT(*) as set_count,
MIN(num_parts) AS min_parts, MAX(num_parts)
AS max_parts, AVG(num_parts) AS avg_parts
FROM sets_with_themes
GROUP BY theme_name;
```

Preprocessing, validation, and joins with SQL

Validation: HAVING

DeepLearning.AI

Validation: HAVING

- Filtering in SQL can happen either before or after grouping
- To filter based on values in individual rows **before** grouping:
  - Use **`WHERE`** statement

```
SELECT *
FROM sets_with_themes
WHERE num_parts > 10;
```

- You may be interested in filtering **after** grouping
- Example:
  - Filter out themes that have fewer than 50 total sets
  - Needs to happen after grouping
  - Won't know number of sets in a group until then

Recap: HAVING

- To filter rows **after** grouping based on the values of aggregation functions

```
SELECT theme_name, COUNT(theme_id) as set_count,
AVG(num_parts) AS avg_parts
FROM sets_with_themes
GROUP BY theme_name
HAVING COUNT(theme_id) > 10
ORDER BY avg_parts DESC;
```

- Common to filter based on aggregations like count and average

Preprocessing, validation, and joins with SQL

Introduction to joins

DeepLearning.AI

Why joins are necessary

Stored across multiple tables:

- ✅ Data organized and efficient
- ⛔ Doesn’t answer questions that require combining information
- ⛔ Have to match and merge data in a time-consuming process

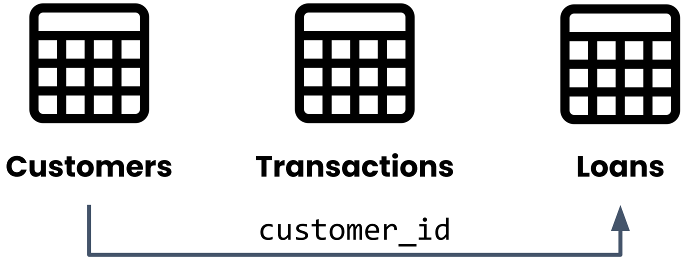
SQL joins:

- ✅ Explicitly link tables using relationships between columns
- ✅ Pull all relevant data together

Example: Peer-to-peer lending product

📊 **Dataset**: Loans from Lending Tree

🎯 **Task**: To know what loans a specific customer took out last year



✅ Use a unique identifier to join the tables together

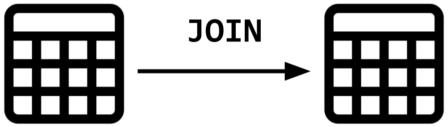
Connecting data with joins

XLOOKUP

- Search one sheet for specific value
- Return corresponding details from another sheet

Joins:

- Search one table for matches
- Bring matches into original table



Customers **Transactions**

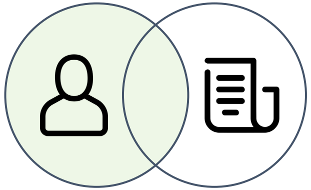
✅ Segment loans by occupation or any attribute from the Customers table.

- Without joins:
  - ⛔ Unscalable
  - ⛔ Nearly impossible to answer complex business questions

Type of JOINs

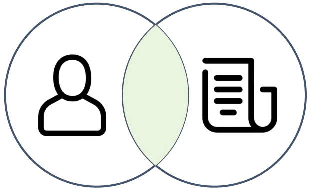
Each join serves a different purpose, depending on how you want to match rows between tables.

LEFT JOIN



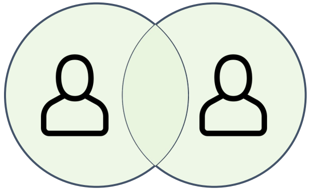
- All rows from left table
- Matching rows from right

INNER JOIN



- Only rows where there is a match in both tables

OUTER JOIN



- All rows from both, even if there isn't a match

Preprocessing, validation, and joins with SQL

Left joins

DeepLearning.AI

Scenario

You
Data Analyst

- You worked with a single table from the LEGO sets database: `sets_with_themes`
    - Table was assembled from multiple tables
- In practice:
    - Different parts of a database are often stored separately
- In case of LEGO database:
    - Sets and themes each have their own table.

Example

Task: Interested in looking at the parts contained in each set

- **LEFT** join to add information about the set each part is contained in


You
Data Analyst

sets_parts
set_num
part_num
color_id
quantity
is_spare

sets_with_themes
set_num
name
year
num_parts
theme_id
theme_name
parent_theme_id
set_num
part_num
color_id
quantity
is_spare

Recap: Left joins

- Used a `LEFT` join to add information about each part to the sets table:
  - `LEFT JOIN` - indicates which table to add to the table in the `FROM` statement
  - `ON` - tells SQL which pair of keys can be used to match rows

```sql
SELECT *
FROM sets_with_themes
LEFT JOIN sets_parts
ON sets_with_themes.set_num = sets_parts.set_num;
```

Add to table

Columns to match

Preprocessing, validation, and joins with SQL

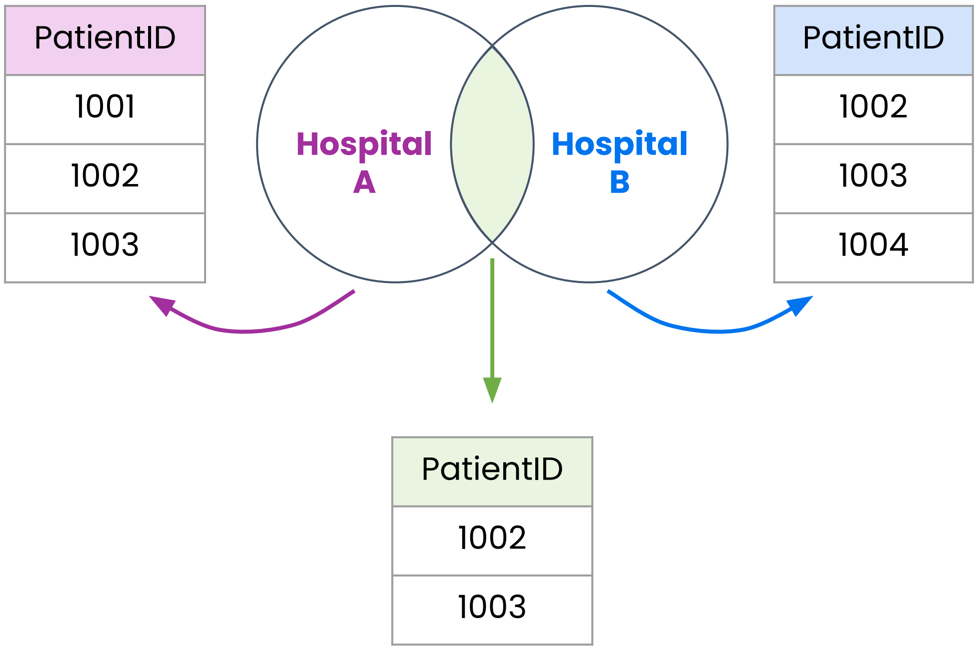
Inner joins

DeepLearning.AI

Inner join

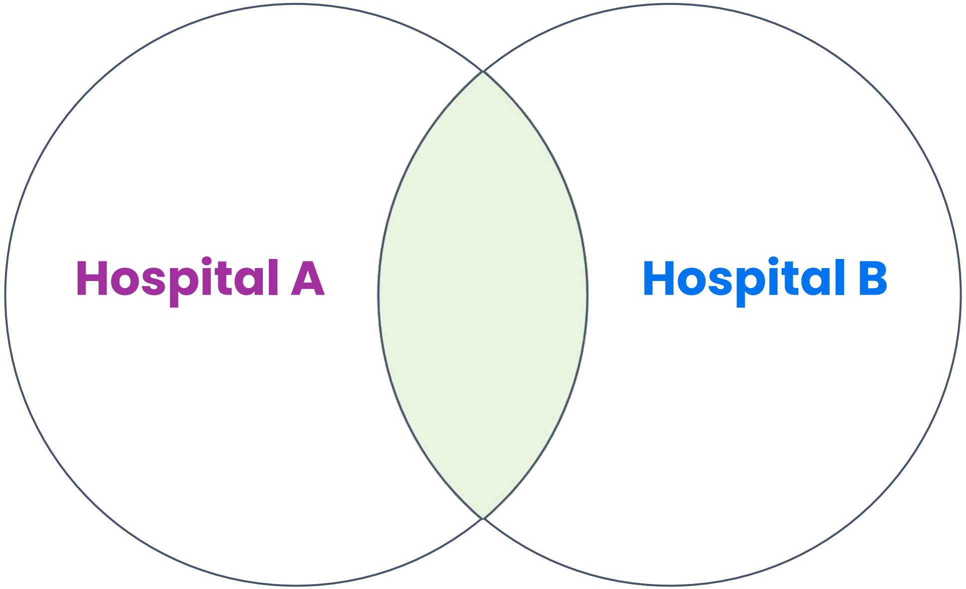
- Combines rows from two tables where:
  - Matching values in columns
  - Specified in the `ON` clause
- Only includes rows present in both tables
- Tables can represent:
  - Same entity
  - Two distinct entities that have a meaningful relationship

```
SELECT *
FROM hospital_A
INNER JOIN hospital_B
ON hospital_A.PatientID = hospital_B.PatientID;
```



Inner join

- Represents the overlapping area where circles intersect
- Patients not included:
  ⛔ Unique to Hospital A
  ⛔ Unique to Hospital B
- **Best for**:
  - Focusing on shared data between two tables.



Intermediate tables

- Intermediate table acts as a bridge
- Intermediate table is essential for many-to-many relationships
  - **Example**: Students and Courses
    - One student: many courses
    - One course: many students

```sql
SELECT Students.Name, Courses.CourseName

FROM Students

INNER JOIN Enrollments ON Students.StudentID = Enrollments.StudentID

INNER JOIN Courses ON Enrollments.CourseID = Courses.CourseID;
```

Students

StudentID	Name	DoB	Email

Enrollments

StudentID	CourseID

Courses

CourseID	CourseName	Department	Instructor

Preprocessing, validation, and joins with SQL

Outer joins

DeepLearning.AI

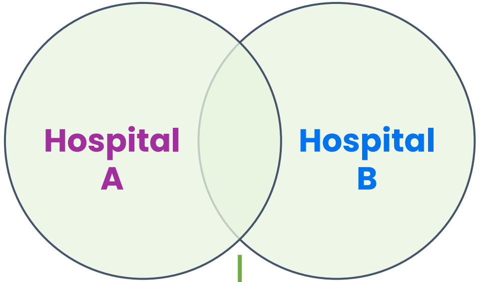
Outer join

- Assembling tables that represent the same entity
- SQLite does not support, but many relational database systems do
- Includes rows from:

  ✅ One or both tables, even if there isn't a match in the other table

```
SELECT *
FROM hospital_A
FULL OUTER JOIN hospital_B
ON hospital_A.PatientID = hospital_B.PatientID;
```

PatientID
1001
1002
1003



PatientID
1002
1003
1004

PatientID
1001
1002
1003
1004

Visualizing outer joins

- Represents the entire area
- All patients are included:
  ✅ Hospital A
  ✅ Hospital B
  ✅ Both
- **Best for**:
  - Comprehensive datasets where you need all records regardless of overlap