SQL Databases and Pandas for AI Workloads

Databases

Pierre Senellart

PSL UNIVERSITÉ PARIS

11 May 2026

Outline

So far in this course

- We have built up a strong toolbox for managing data:
  - the relational model and integrity constraints
  - relational algebra and SQL
  - relational calculus and Codd's theorem
  - indexing, query optimization, transactions
  - Python programs talking to a DBMS
- Everything has been organized around a database management system (PostgreSQL, MySQL, SQLite...)
- A natural question arises:

Do we always need a DBMS?

When the DBMS is necessary

A DBMS earns its complexity when you need:

- concurrent access from many users or applications
- durability and transactions (ACID guarantees)
- integrity constraints enforced by the system
- a declarative query language with automatic optimization
- handling very large data, with indexes, beyond main memory
- data shared across independent applications

When the DBMS is overkill

But many real workloads do not need any of that:

- the dataset fits in memory (a few GB on a laptop)
- there is a single user (a data scientist, a script)
- the data is read-only or append-only
- the query is a precise sequence of transformations rather than a high-level question
- we want tight integration with code (plotting, machine learning, deep learning libraries)

When the DBMS is overkill

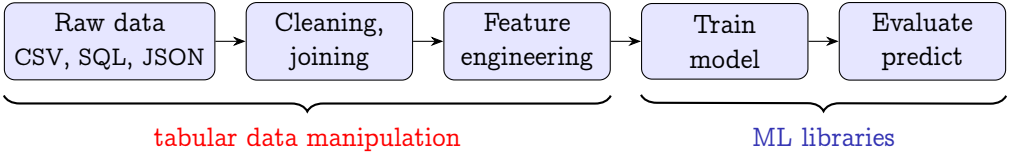
But many real workloads do not need any of that:

- the dataset fits in memory (a few GB on a laptop)
- there is a single user (a data scientist, a script)
- the data is read-only or append-only
- the query is a precise sequence of transformations rather than a high-level question
- we want tight integration with code (plotting, machine learning, deep learning libraries)

This is exactly the situation during the in-memory feature-engineering stage of most AI / data science pipelines.

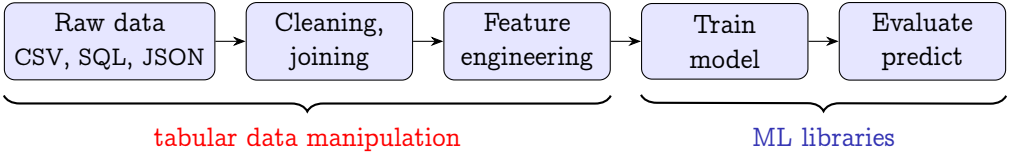
The AI pipeline

A typical machine-learning workflow:



The AI pipeline

A typical machine-learning workflow:



In Python, the standard tool for the left side is pandas.

Outline

Pandas

- Python library for tabular data manipulation
- Central object: the `DataFrame`
- Heavily inspired by tabular data handling in R and SAS
- De facto standard in data science: every Python ML tutorial, every Kaggle notebook, every pandas-based pipeline
- Tight integration with the rest of the scientific Python ecosystem:
  - NumPy (numerical arrays)
  - Matplotlib, Seaborn (plotting)
  - scikit-learn, PyTorch, TensorFlow, XGBoost (machine learning)
- Documentation: `https://pandas.pydata.org/docs/`

Pandas vs SQL: different beasts

- SQL is a declarative query language:
  - you describe what you want
  - the DBMS decides how to compute it
- Pandas is an imperative library:
  - you describe how to compute the result, step by step
  - the order of operations is the order of execution
  - only minor automatic optimizations
  - consequence: SQL benefits from decades of query optimization research and from the DBMS's indexes, plans, and statistics; pandas does none of this for you
- Pandas code is often less verbose than SQL...

Pandas vs SQL: different beasts

- SQL is a declarative query language:
  - you describe what you want
  - the DBMS decides how to compute it
- Pandas is an imperative library:
  - you describe how to compute the result, step by step
  - the order of operations is the order of execution
  - only minor automatic optimizations
  - consequence: SQL benefits from decades of query optimization research and from the DBMS's indexes, plans, and statistics; pandas does none of this for you
- Pandas code is often less verbose than SQL... but also more cryptic
- All data is in main memory: pandas does not scale to terabytes
- But: arbitrary Python is at hand: loops, functions, libraries

The story of this lecture

Key idea

Pandas may look like a different world, but its core data manipulation primitives are essentially the operators of the relational algebra you already know.

We will:

- introduce the DataFrame object
- translate each relational algebra operator into pandas
- see how to bridge SQL and pandas in a real AI pipeline
- discuss when to use which

Outline

The DataFrame object

- Representation of a relation: tabular data, fixed number of named columns
- Each row can be assigned an index, similar to a primary key; if none is given, rows get a default integer numbering
- Each column is a `Series`, a one-dimensional array with a name and a dtype
- A `DataFrame` has two axes:
  - axis 0: the rows
  - axis 1: the columns
- Columns also have column indexes (their names)

Constructing a DataFrame

Python

```python
import pandas as pd

# Literal DataFrame
guest = pd.DataFrame(
  data={
    'name':  ['John Smith', 'Alice Black', 'John Smith'],
    'email': ['john.smith@gmail.com',
              'alice@black.name',
              'john.smith@ens.fr']
  },
  index=pd.Index([1, 2, 3], name='id'))

# Read from a CSV file, first column as the row index
reservation = pd.read_csv('reservation.csv', index_col=0)
```

We will see how to read from an SQL database later in the lecture.

Our running example

We reuse the schema from the SQL lectures.

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

Outline

Renaming the index

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\rho_{\texttt{id}\to\texttt{guest}}(\texttt{Guest})$$

Python

```
guest.index.name = 'guest'
```

(modifies the guest DataFrame in place)

Renaming a column

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\rho_{\texttt{email}\to\texttt{e-mail}}(\texttt{Guest})$$

Python

```
guest.rename(columns={'email': 'e-mail'})
```

Projection

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\Pi_{\texttt{email},\texttt{id}}(\texttt{Guest})$$

Python

```
guest[['email']]
```

(the row index always comes along; here `id` is the index)

Projection (removing the index)

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\Pi_{\texttt{email}}(\texttt{Guest})$$

Python

```
guest.reset_index()[['email']]
```

(a fresh default integer index is generated)

Python

```
guest.set_index('email')[[]]
```

(or promote `email` to be the new index)

Selection

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\sigma_{\texttt{arrival}>\texttt{2026-01-12}\wedge\texttt{guest}=2}(\texttt{Reservation})$$

Python

```
reservation[(reservation.arrival > '2026-01-12') &
            (reservation['guest'] == 2)]
```

(see next slide for what goes on inside the brackets)

Selection: boolean indexing, step by step

Each comparison produces a column of `True`/`False` values, one entry per row of `reservation`:

Python

```
reservation.arrival > '2026-01-12'  # column [F, F, T, T, T]
reservation['guest'] == 2           # column [F, T, F, T, T]
```

- `&` combines two such columns element-wise with a logical $\wedge$: the result here is `[F, F, F, T, T]`
- Python's `and` works on a single Boolean and cannot be used (same for `|` as $\vee$ in lieu of Python's `or` and `~` as $\neg$ in lieu of Python's `not`)
- parentheses around each comparison are mandatory: `&` binds tighter than `>` and `==`
- indexing the `DataFrame` with the resulting Boolean column keeps the rows whose entry is `True`

Outline

Cross product

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\Pi_{\texttt{id}}(\texttt{Guest}) \times \Pi_{\texttt{name}}(\texttt{Guest})$$

Python

```
guest[[]].reset_index().merge(
    guest[['name']], how='cross'
).drop_duplicates().sort_values(['name','id'])
```

Union

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2}(\texttt{Reservation}))$$
$$\cup\, \Pi_{\texttt{room}}(\sigma_{\texttt{arrival}=\texttt{2026-01-15}}(\texttt{Reservation}))$$

Python

```
pd.concat([
  reservation[reservation.guest == 2]
    .reset_index()[['room']],
  reservation[reservation.arrival == '2026-01-15']
    .reset_index()[['room']]
]).drop_duplicates()
```

Difference

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\Pi_{\texttt{room}}(\sigma_{\texttt{guest}=2}(\texttt{Reservation}))$$
$$\setminus\ \Pi_{\texttt{room}}(\sigma_{\texttt{arrival}=\texttt{2026-01-15}}(\texttt{Reservation}))$$

Python

```
r1 = reservation[reservation.guest == 2] \
       .reset_index()[['room']]
r2 = reservation[reservation.arrival == '2026-01-15'] \
       .reset_index()[['room']]
r1.merge(r2, how='outer', indicator=True) \
  .query('_merge == "left_only"')[['room']] \
  .drop_duplicates()
```

Join

Guest

id	name	email
1	John Smith	john.smith@gmail.com
2	Alice Black	alice@black.name
3	John Smith	john.smith@ens.fr

Reservation

id	guest	room	arrival	nights
1	1	504	2026-01-01	5
2	2	107	2026-01-10	3
3	3	302	2026-01-15	6
4	2	504	2026-01-15	2
5	2	107	2026-01-30	1

$$\texttt{Reservation} \bowtie_{\texttt{guest}=\texttt{id}} \texttt{Guest}$$

Python

```
pd.merge(reservation, guest,
         left_on='guest', right_on='id')
```

(`how='left'`, `'right'`, `'outer'` for the corresponding outer joins)

Outline

Aggregation with groupby

Average number of nights per room, keeping only rooms with average > 3 (recall: aggregation is beyond the classical relational algebra):

SQL

```sql
SELECT room, AVG(nights) AS avg
FROM reservation
GROUP BY room
HAVING AVG(nights) > 3
ORDER BY room;
```

Python

```python
reservation.groupby('room')[['nights']] \
    .mean() \
    .query('nights > 3') \
    .sort_values(by='room')
```

Summary: the RA → pandas dictionary

Relational algebra	SQL	Pandas
$\rho_{A \to B}$	`AS`	`df.rename(columns=...)`
$\Pi_{A_1,\ldots,A_n}$	`SELECT`	`df[[...]]`
σ_φ	`WHERE`	`df[mask]`
$R \times S$	`CROSS JOIN`	`R.merge(S, how='cross')`
$R \cup S$	`UNION`	`pd.concat([R,S]).drop_duplicates()`
$R \setminus S$	`EXCEPT`	`merge(..., indicator=True)...`
$R \bowtie_\varphi S$	`JOIN ... ON`	`pd.merge(R, S, on=..., how=...)`
(aggregation)	`GROUP BY`	`df.groupby(...).agg(...)`

Once you see the mapping, every relational query has a mechanical translation to pandas.

Outline

Convenient pandas idioms

`df.sort_values` order results (`ORDER BY`)

`df.head(n)` / `df.tail(n)` first / last rows (`LIMIT`)

`df.shape`, `df.size` dimensions / number of cells

`df.describe()` summary statistics per numeric column

`df.apply(f)` apply an arbitrary Python function row- or column-wise

Things SQL cannot easily do

- Plot a column in two lines:

Python

```
import matplotlib.pyplot as plt
reservation['nights'].plot(kind='hist'); plt.show()
```

- Train a model on a DataFrame:

Python

```
from sklearn.linear_model import LinearRegression
X = features[['nights', 'room']]
y = features['price']
LinearRegression().fit(X, y)
```

- Use any Python library on each row, with full programming power
- This is why pandas dominates the feature-engineering and modelling stages: it lives inside Python

Things pandas cannot easily do

- Enforce integrity: no `NOT NULL`, no `UNIQUE`, no `FOREIGN KEY` – a typo in a join key silently produces an empty result, not an error
- Use an index: every `df[df.x == v]` is a linear scan; in MySQL with a B^+-tree, the same lookup is logarithmic
- Handle data that does not fit in RAM: pandas loads everything; the DBMS streams from disk and uses buffers
- Serve concurrent users: no transactions, no isolation – two scripts editing the same DataFrame race silently
- Optimize a query for you: pandas executes operations in the order you wrote them; the DBMS reorders joins, pushes filters down, and picks indexes
- Persist across processes and machines, with backups, replication, point-in-time recovery

This is why pandas does not replace the DBMS in production but complements it.

Outline

From SQL to a DataFrame

Pandas ships with an SQL bridge, on top of SQLAlchemy:

Python

```
import pandas as pd
from sqlalchemy import create_engine
engine = create_engine(
  'mysql+mysqlconnector://student:s3cr3t@localhost/hotel')

df = pd.read_sql_query("""
    SELECT g.name, COUNT(*) AS n_bookings, SUM(r.nights) AS total
    FROM guest AS g JOIN reservation AS r ON g.id = r.guest
    GROUP BY g.id, g.name
    HAVING COUNT(*) > 1
""", engine)
```

From SQL to a DataFrame

Pandas ships with an SQL bridge, on top of SQLAlchemy:

Python

```
import pandas as pd
from sqlalchemy import create_engine
engine = create_engine(
  'mysql+mysqlconnector://student:s3cr3t@localhost/hotel')

df = pd.read_sql_query("""
    SELECT g.name, COUNT(*) AS n_bookings, SUM(r.nights) AS total
    FROM guest AS g JOIN reservation AS r ON g.id = r.guest
    GROUP BY g.id, g.name
    HAVING COUNT(*) > 1
""", engine)
```

- Heavy lifting (join, group by, filter) delegated to the DBMS, which uses indexes and optimization
- Result returned as a `DataFrame` ready for analysis or training

From a DataFrame to SQL

And the other direction:

Python

```
df.to_sql('predictions', engine,
          if_exists='replace', index=False)
```

Use cases:

- write back model predictions or cleaned data
- materialize a snapshot for downstream applications
- share results with other users via the DBMS

Outline

When to use which?

	SQL / DBMS	Pandas
Concurrent users, transactions	yes	no
Persistent storage, durability	yes	no
Integrity constraints	yes	no
Indexed lookups on huge tables	yes	no
Data does not fit in RAM	yes	no
In-memory feature engineering	no	yes
Tight integration with sklearn/PyTorch	no	yes
Plotting, ad-hoc exploration	no	yes
Iterative notebook workflow	no	yes
Arbitrary Python on each row	no	yes

A realistic AI pipeline

1. Storage: raw data lives in a relational database (logs, transactions, sensors...)
2. Extraction: an SQL query, using joins, filters, aggregations, returns a small enough working set
3. Bridge: `pd.read_sql_query` brings it into a DataFrame
4. Feature engineering: pandas, numpy, custom Python functions
5. Training: scikit-learn, PyTorch, XGBoost...
6. Persistence: predictions are written back via `df.to_sql`

A realistic AI pipeline

1. Storage: raw data lives in a relational database (logs, transactions, sensors...)
2. Extraction: an SQL query, using joins, filters, aggregations, returns a small enough working set
3. Bridge: `pd.read_sql_query` brings it into a DataFrame
4. Feature engineering: pandas, numpy, custom Python functions
5. Training: scikit-learn, PyTorch, XGBoost...
6. Persistence: predictions are written back via `df.to_sql`

Each step uses the right tool, and you have now seen both ends.

Beyond pandas

pandas has limits; if you hit them, look at:

- Polars: pandas-like API, much faster, lazy execution
- DuckDB: in-process SQL engine over DataFrames, columnar
- Dask, Ray: distributed pandas-like computing
- Apache Arrow: in-memory columnar exchange format

But the conceptual model you have learned (the relational algebra) carries over to all of them.

Take-aways

- In practice, the right answer is SQL and pandas together, not SQL or pandas: SQL where it shines (storage, heavy queries, integrity), pandas where it shines (in-memory feature engineering, ML integration)
- The core of pandas is essentially the relational algebra you already know – everything you learned about SQL transfers directly
- pandas is the lingua franca of AI data prep *inside Python*; the DBMS remains the durable home of the data
- Each tool's strength is the other's weakness: keep both in your toolbox

License