Importing Data into Python from Relational Databases

Xavier Morera
BIG DATA INC.

@xmorera www.xaviermorera.com

Importing sqlite Database Files

Database engine

- Store & work with relational data
- Simple to use and portable

Library and the db file

- Use sqlite3 in Python

Many notable users

```
import sqlite3
# ls
stack_connection = sqlite3.connect('importing_sqlite.db')
type(stack_connection)
stack_cursor = stack_connection.cursor()
stack_cursor.execute("select name from sqlite_master where type = 'table';")
stack_cursor.fetchone()
```

Importing sqlite Database Files

Import sqlite3 and create a connection using connect

Create a Cursor object, and execute

Retrieve results using fetchone or fetchall

```
# ls -l

stack_connection_bad = sqlite3.connect('bad_name_sqlite.db')
stack_connection_bad.cursor().execute("select name from sqlite_master where type =
'table';").fetchall()

# ls -l
```

Watch Out!

Make a mistake on the database name

And a new, empty database may be created

```
rows = stack_cursor.execute('select * from posts').fetchall()
type(rows)
rows[0]
type(rows[0])
stack_cursor.execute('select * from posts limit 1').fetchall()
stack_cursor.execute('select Id, Score, Tags from posts limit 3').fetchall()
```

Querying Data

Explore your data

- Returns a list of objects of type tuple

Refine your queries

Table of Contents

`sqlite3` — DB-API 2.0 interface for SQLite databases

Source code: Lib/sqlite3/

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")
```

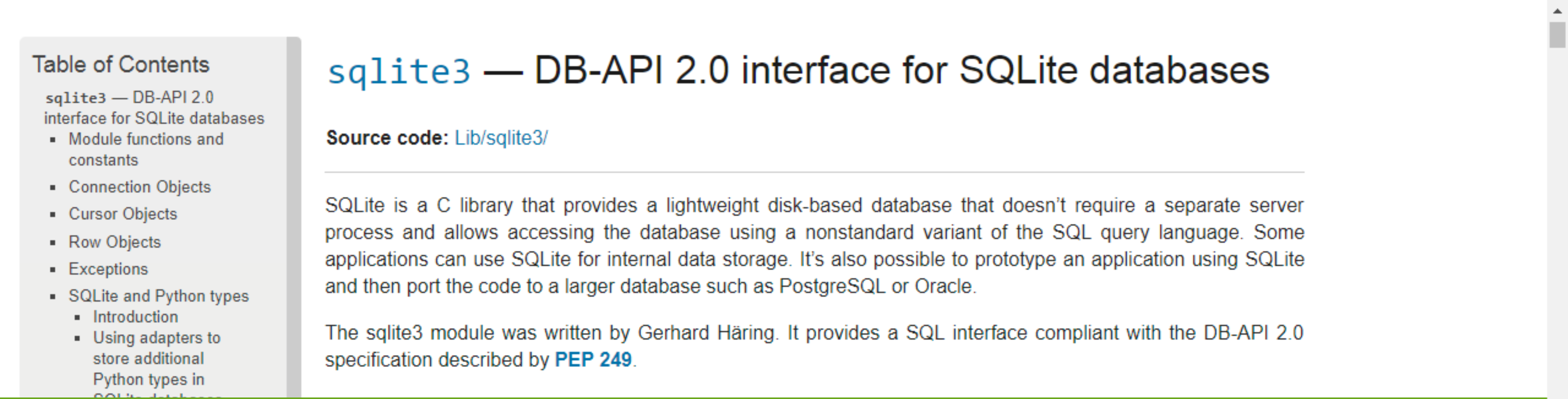
```
import pandas as pd

stack_connection = sqlite3.connect('importing_sqlite.db')

posts_df = pd.read_sql("select * from posts;", stack_connection)

type(posts_df)

posts_df.columns

posts_df.head()
```

Taking Advantage of Pandas

Working with a list of tuples may not be ideal

Enter pandas

- Still with sqlite3, but you create a DataFrame
- Use read_sql

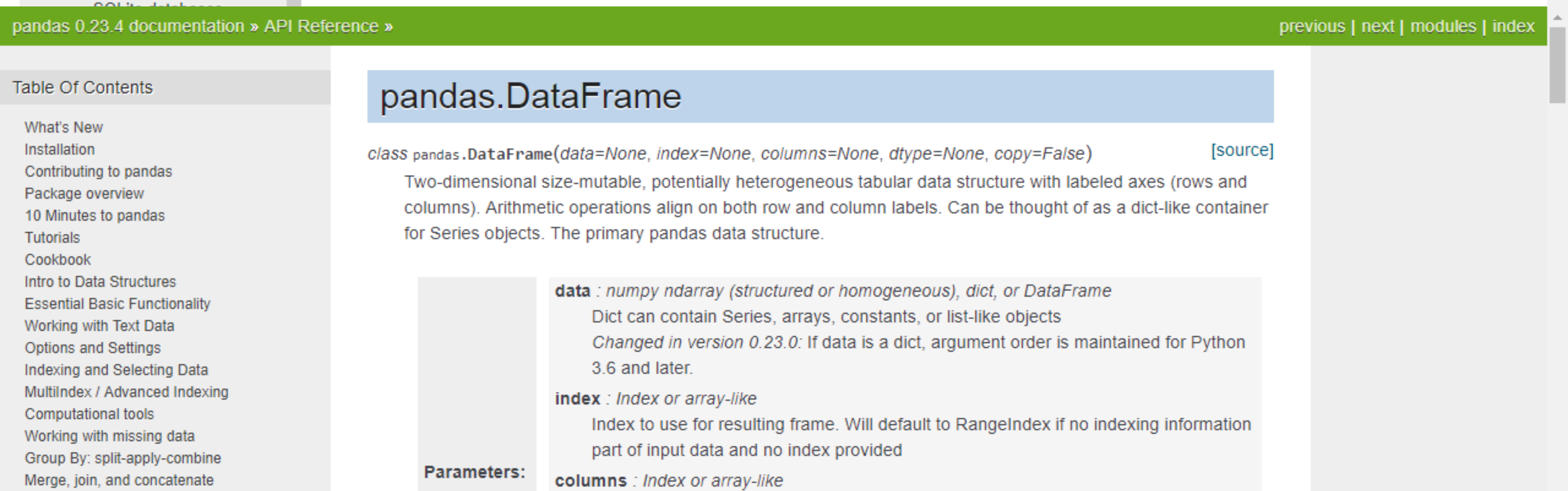


sqlite3 — DB-API 2.0 interface for SQLite databases

Source code: Lib/sqlite3/

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**.



pandas.DataFrame

class `pandas.DataFrame`(*data=None, index=None, columns=None, dtype=None, copy=False*) [source]

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure.

Parameters:

data : *numpy ndarray (structured or homogeneous), dict, or DataFrame*
Dict can contain Series, arrays, constants, or list-like objects
Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

index : *Index or array-like*
Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided

columns : *Index or array-like*

Working with Databases Using SQLAlchemy

SQLAlchemy

The Python SQL toolkit

Supports

- SQLite, PostgreSQL, MySQL
- Oracle, MS SQL, Firebird, Sybase…

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///importing_sqlite.db')
type(engine)
dir(engine)
engine.table_names()
engine.url
engine.dialect
engine.driver
```

Using SQLAlchemy with SQLite

SQLAlchemy

Start with the necessary import

And then create_engine

- Provide connection string
- You have an engine

Anatomy of a Connection String

```
engine = create_engine('sqlite:///importing_sqlite.db')
```

Anatomy of a Connection String

```
engine = create_engine('sqlite:///importing_sqlite.db')
```

Anatomy of a Connection String

sqlite:///importing_sqlite.db

Anatomy of a Connection String

dialect:///importing_sqlite.db

Anatomy of a Connection String

dialect:///dbname

Anatomy of a Connection String

dialect[+driver]://user:password@hostname/dbname

Anatomy of a Connection String

```
dialect[+driver]://user:password@hostname/dbname[?key=value]
```

```
engine_sqlite = create_engine('sqlite:///importing_sqlite.db')

engine_mysql = create_engine('postgresql://xavier:postgres@localhost:5432/importing_postgres')

engine_postgresql = create_engine('mysql+mysqlconnector://root:mysql@localhost:3306/importing_mysql')
```

Connecting to Your Database of Choice

Layer of abstraction

Create database agnostic applications

Load into a Pandas DataFrame

```
engine_sqlite = create_engine('sqlite:///importing_sqlite.db')

engine_postgres = create_engine('postgresql://xavier:postgres@localhost:5432/importing_postgres')

engine_mysql = create_engine('mysql+mysqlconnector://root:mysql@localhost:3306/importing_mysql')
```

MySQL™

Open source database

Runs on virtually all major platforms

- Top 3 of most widely used

Client-server model

```
# show databases;
# use importing_mysql
# show tables;
engine = create_engine('mysql+mysqlconnector://root:mysql@localhost:3306/importing_mysql')
posts = pd.read_sql_table('posts', engine, index_col='Id')
type(posts)
posts.columns
posts.head()
```

Importing Data with Pandas

Use SQLAlchemy to connect to MySQL with pandas

- Create engine
- Requires mysql-connector-python

Load table into DataFrame with read_sql_table and set index column

pandas.read_sql_table

`pandas.read_sql_table`(*table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None*) [source]

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

Parameters:

table_name : *string*
Name of SQL table in database.

con : *SQLAlchemy connectable (or database string URI)*
SQLite DBAPI connection mode not supported.

schema : *string, default None*
Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

index_col : *string or list of strings, optional, default: None*
Column(s) to set as index(MultiIndex).

coerce_float : *boolean, default True*
Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.

parse_dates : *list or dict, default: None*

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

columns : *list, default: None*

```
posts = pd.read_sql_table('posts', engine, columns=['Id', 'CreationDate', 'Tags'])
posts.head()
type(posts.iloc(1)[1])
posts = pd.read_sql_table('posts', engine, columns=['Id', 'CreationDate', 'Tags'],
parse_dates={'CreationDate': {'format': '%Y-%m-%dT%H:%M:%S.%f'}})
type(posts.iloc(1)[1])
```

Importing Data with Pandas

Additional functionality

- Select only specific columns
- Use parse_dates

pandas.read_sql_table

`pandas.read_sql_table`(*table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None*) [source]

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

Parameters:

table_name : *string*
Name of SQL table in database.

con : *SQLAlchemy connectable (or database string URI)*
SQLite DBAPI connection mode not supported.

schema : *string, default None*
Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

index_col : *string or list of strings, optional, default: None*
Column(s) to set as index(MultiIndex).

coerce_float : *boolean, default True*
Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.

parse_dates : *list or dict, default: None*

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

columns : *list, default: None*

PostgreSQL

+

Psycopg2

PostgreSQL is a solid database

Psycopg2

- Popular database adapter
- PostgreSQL + Python

```
# psql
# \l
# \c importing_postgres
# \d
# SELECT "Id", "Title" FROM posts LIMIT 5;
import psycopg2
stack_connection = psycopg2.connect("dbname=importing_postgres user=xavier
host=localhost")
so_cursor = stack_connection.cursor()
```

Importing Data with Psycopg2

Start by importing psycopg2

Create a connection using connect

Create a cursor

```
so_cursor.execute("select * from posts")
first_row = so_cursor.fetchone()
first_row
type(first_row)
rows = so_cursor.fetchall()
rows
type(rows)
```

Importing Data with Psycopg2

Query with execute using the cursor

Get your results

- Use fetchone or fetchall

```
stack_connection.commit()
stack_connection.close()
```

Importing Data with Psycopg2

Remember to commit when not using the connection

And close when connection no longer needed

```
engine_mysql.table_names()

nicer_query = "SELECT posts.Id, Users.DisplayName, posts.AnswerCount,
posts.ViewCount FROM posts INNER JOIN users on
posts.OwnerUserId=Users.Id ORDER BY posts.ViewCount DESC LIMIT 5;"

posts = pd.read_sql(nicer_query, engine_mysql)
```

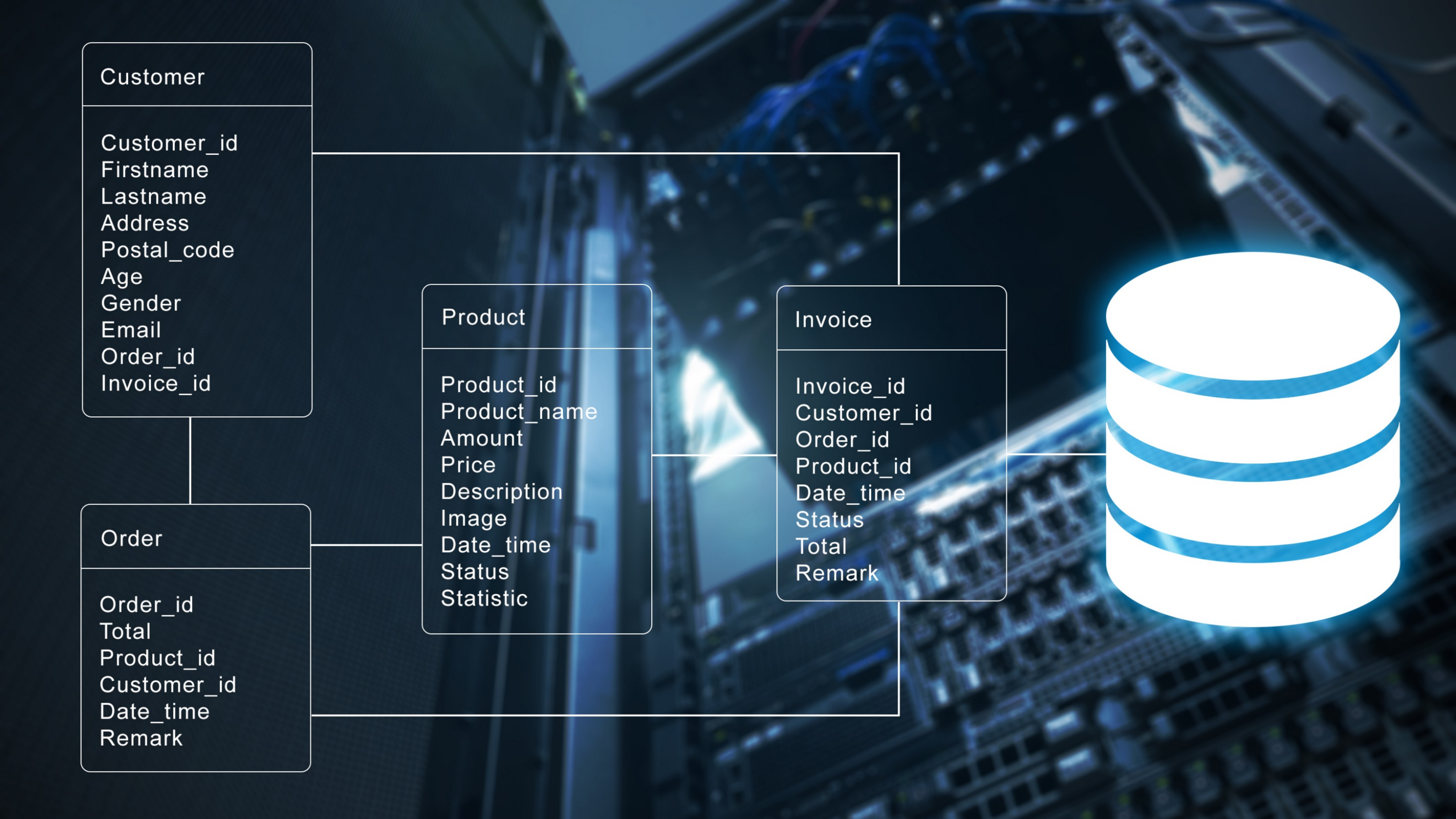
Importing Relational Data

SQL available

- Use it to its full potential

Complex queries, and load into DataFrame

- With read_sql


Customer
Customer_id
Firstname
Lastname
Address
Postal_code
Age
Gender
Email
Order_id
Invoice_id
Product
Product_id
Product_name
Amount
Price
Description
Image
Date_time
Status
Statistic
Invoice
Invoice_id
Customer_id
Order_id
Product_id
Date_time
Status
Total
Remark
Order
Order_id
Total
Product_id
Customer_id
Date_time
Remark

Importing Data: Python Data Playbook

Xavier Morera
BIG DATA INC.

@xmorera www.xaviermorera.com