PATSTAT SQL for Python v1.0

Quick start guide

epo.org

Prepared by the European Patent Office
EPO Observatory on Patents and Technology

Version 1.0
Last updated: 27 May 2026

This document serves as a comprehensive introduction and practical guide for beginners to PATSTAT, the European Patent Office's global patent statistics database. It covers foundational concepts of relational databases, detailed SQL query construction and practical examples tailored to accessing and analysing patent data within PATSTAT, especially using the Technology Intelligence Platform (TIP) environment.

The content of this quick-start guide is intended for training and information purposes only. The information it contains is of a general nature only and is not intended to address the specific circumstances of any particular case, individual or entity.

The EPO cannot guarantee that the information is comprehensive, complete, accurate or up-to-date, and they therefore cannot accept responsibility for any loss or damage that may arise from reliance on it.

The information contained in this quick-start guide in no way constitutes professional or legal advice.

Table of contents

Glossary

CPC	Cooperative Patent Classification (CPC): a patent classification system developed in partnership between the USPTO and the EPO. See also A2.3.
DOCDB	The EPO's master documentation database with worldwide coverage. It contains bibliographic data, abstracts, citations and DOCDB simple patent family information. See also A2.2.
Espacenet	Free online patent searching service developed by the EPO. It includes information on over 160 million patent documents from more than 100 patent offices on all continents. Espacenet is available at epo.org/espacenet.
First filing	First patent application filed for a new invention, typically with a national or regional patent office.
INPADOC	EPO worldwide legal event data. INPADOC bulk data is the backbone of many EPO legal status products and services. It includes legal status events from over 40 international patent authorities worldwide and is available as backfile and frontfile data. See also A2.2.
IPC	The International Patent Classification (IPC) system is a hierarchical patent classification system used by the EPO and more than 100 patent offices worldwide. It breaks technologies down into eight sections with several hierarchical sub-levels. The IPC system has more than 70 000 subdivisions and is updated on an annual basis. See also A2.3.
Jupyter Notebook	Jupyter Notebooks are a community standard for communicating and performing interactive computing. They are used to blend computations, outputs, explanatory text, mathematics, images and rich media representations of objects.
ORM	Object-relational mapping (ORM) is a software technique that links programming objects to the data stored in relational databases. It resolves object-oriented code with database tables, allowing developers to work with data as objects while abstracting underlying database structures. This enables applications to remain modular and largely independent of changes to the database design.
Patent classification system	The set of patent classification symbols assigned to categorise the technical subject-matter of a patent or utility model. There are various patent classification systems used today by national, regional and international patent offices. See also A2.3.

Patent family	A set of patent documents covering the same or similar technical content, depending on the patent family definition. See also A2.2.
PATSTAT	PATSTAT is a group of databases that contain bibliographical, procedural and other context information on millions of patents and utility models from numerous industrialised and developing countries. It is built from the EPO's databases of worldwide patent data. See also A1.1.
PCT	Patent Cooperation Treaty (PCT): an international treaty providing for a unified procedure for filing patent applications to protect inventions in its contracting states. Under the PCT, a single international application can be filed for patent protection in up to more than 150 countries. The PCT provides for a centralised procedure for filing the patent application whereby the substantive examination and the grant of the patent lie with the competent national or regional patent office(s).
Priority	Inventions can be protected by patents and utility models in more than one country. For a period of 12 months from the date of filing an application for a patent in a member state of the Paris Convention, the applicant or their successor can claim a right of priority from that application for any subsequently filed patent application that concerns the same invention. If the requirements are fulfilled, the date of the earlier application counts as the date of filing of the later application for the purposes of examining novelty and inventive step.
Python	Python is a high-level, interpreted, object-oriented programming language known for its clear syntax and readability. It supports rapid application development, scripting, and modular programming through extensive built-in data structures, libraries, and package support.
SQL	Structured Query Language: a database sublanguage used in querying, updating and managing relational databases – the de facto standard for database products.
TIP	Technology Intelligence Platform (TIP): a new EPO data analysis tool offering a JupyterLab-based environment featuring a powerful computing infrastructure and providing easy access to patent data.

PATSTAT for beginners

The purpose of this quick-start guide is to introduce users to the general concepts underlying PATSTAT, the European Patent Office's global patent statistics database, and to provide an initial orientation on how to start working with the data. It covers not only high-level aspects of the database architecture, but also introductory programming concepts that enable users to interact with the EPO's Technology Intelligence Platform (TIP) and perform practical operations.
Historically, PATSTAT workflows have been largely SQLcentric, i.e. based on Structured Query Language (SQL). Looking ahead, Python object-relational mapping (ORM) objects are intended to complement SQL, enhancing composability, reusability and integration with modern analytics stacks. For this reason, this document and the Quick-start guide to PATSTAT SQL contain the same content presented in two different programming languages. This document **focuses on Python ORM language**, while the Quick-start guide to PATSTAT SQL provides equivalent concepts implemented in SQL (to be used, for example, on TIP). TIP is a versatile platform, and users are free to choose the programming language that best suits their preferences and workflows.

This quick-start guide is structured as follows: after a brief introduction to the PATSTAT database and its context, it outlines the fundamentals of relational databases, explaining the role of tables, keys and relationships. The central part is devoted to ORM and progressively introduces the structure of queries, the core commands used to select, filter, sort and combine data, as well as the main aggregation techniques and commonly used functions. This is followed by a section offering practical guidance on writing robust queries and an overview of how to access and configure PATSTAT. The final section presents a collection of query examples that includes both quick queries for common use cases and recurring patterns for more advanced analyses.

1. Introduction to PATSTAT

PATSTAT is the European Patent Office's curated database suite for global patent statistics and research, designed to support largescale quantitative analyses of patenting activity across jurisdictions and over time. It is organised as a highly normalised relational database, in which information on applications, publications, applicants, inventors, classifications, citations, families and legal events is distributed across multiple interlinked tables. While this design enables flexible and powerful analyses, it also requires users to combine data from several sources through well-constructed queries.
Thanks to its breadth, consistency and long-time coverage, PATSTAT is widely used for patent statistics, technology intelligence, innovation studies and policy analysis. At the same time, its richness and complexity make it essential for users to understand both the underlying data model and the methodological implications of their query design.
The PATSTAT package consists of two main products: **PATSTAT Global**, which provides worldwide coverage of patent applications and publications, and **PATSTAT EP Register**, which focuses on legal events and the procedural history of European patent applications. Both products are released twice a year, with Spring editions reflecting a snapshot taken in late January, and Autumn editions capturing the state of the data in late July.
When working with PATSTAT data, it is important to consider **inherent publication and processing delays**. Patent applications are typically published 18 months from the date of filing or, if priority has been claimed, from the date of priority, and national data delivery and subsequent harmonisation may lead to additional delays. As a result, users should apply an appropriate safety margin when interpreting the apparent recency of the data, particularly for analyses involving the most recent years.

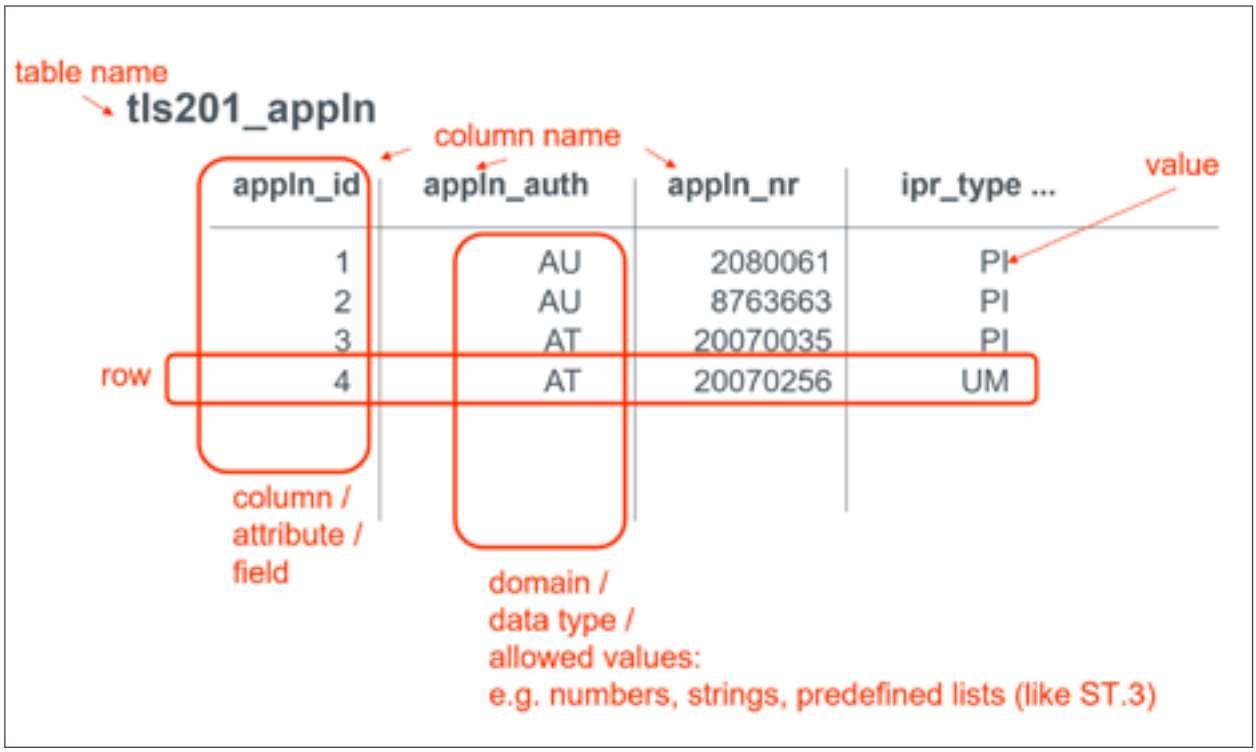
More information on PATSTAT web pages.

2. Relational databases

PASTAT is a relational database, and in the following section we explore what this means.

2.1 Essentials of relational databases

A relational database is a collection of linked tables composed of columns (attributes) and rows (records). Each cell contains a data item of a certain **type**, which can be a string, a number or a date. Relational databases are well suited for structured data, but not for text or images (because e.g. text is usually treated as a complex data element containing paragraphs, sentences, words, etc.).



2.2 Keys and relationships

In a relational database, “keys” are special columns to link to other tables. A **primary key** is a column, or combination of columns, that uniquely identifies each row in a table, ensuring each entry is distinct. A **foreign key** is a column that points to the primary key in another table, creating links between different sets of information. These keys help organise data in useful ways. For example, a **one-to-many relationship** means that one row in a table can be linked to several rows in another table, as in the case of an applicant filing multiple patent applications.

Technically, this is implemented by storing the applicant's primary key (person_id) as a foreign key in the person_application table (tls207_pers_appln), so that multiple rows can reference the same original record. A **many-to-many relationship** means that records in two tables can be associated with multiple records on both sides, such as applications having multiple applicants or inventors. In relational databases, this type of relationship cannot be represented directly, so it is implemented through an intermediate join table, which stores the primary keys of both original tables as foreign keys, allowing each application, applicant or inventor relationship to be recorded explicitly.

PATSTAT follows these general principles in a highly normalised schema. Each core entity is identified by a primary key. For example, patent applications are uniquely identified by the application identifier (*appln_id*) in the applications table (*tls201_appln*), while persons are uniquely identified by the person identifier (*person_id*) in the persons table (*tls206_person*). Application information and person information cannot be joined directly because a single patent application may involve multiple persons, and the same person may be linked to many different applications. To represent this situation, PATSTAT uses an intermediate join table, *tls207_pers_appln*, which stores both the application identifier and the person identifier as foreign keys, so that each row represents a specific association between an application and a person, therefore implementing a many-to-many relationship between applications and persons.

Understanding how these keys and relationships are implemented in PATSTAT is essential for constructing correct joins and for interpreting query results accurately. More information on the data scheme of the databases can be found in the following links for PATSTAT Global and PATSTAT EP Register.

3. Introduction to ORM

ORM provides a programmatic way to interact with relational databases using the concepts and syntax of a general-purpose programming language. It avoids the need to manually write SQL database queries. Through ORM, database tables are represented as Python objects, columns as attributes, and relationships as explicit links between classes. ORM follows a declarative and compositional approach: users describe which data they want by chaining high-level operations (such as selecting attributes, applying conditions, or limiting results), and without needing to specify how the database should execute the query.

In contrast to the different SQL dialects implemented by various database management systems (DBMSs), ORM does not follow a formal standard. Differences instead arise within the ORM framework itself, as each ORM defines its own query API, syntax and abstractions. Even within the same programming language, different ORMs expose different ways of expressing queries, relationships and results. In this guide, we use the PATSTAT Python ORM, which is based on **SQLAlchemy**'s ORM Query interface.

Note before starting

This document is intended to facilitate the use of PATSTAT via TIP. The TIP platform operates within a Jupyter-based environment, providing support for Python along with an extensive selection of libraries and packages. To use SQLAlchemy's ORM within TIP and to view the results, it is essential first to correctly set up the PATSTAT client, and then to assign the result to a dataframe.

```
#Import and initialise the Patstat Client
from epo.tipdata.patstat import PatstatClient
patstat = PatstatClient(env='PROD')
db = patstat.orm()
# Create the query
q_appln = db.query(...) # Input the query here
# Visualise the results
appln = patstat.df(q_appln)
appln.head()
```

3.1 The three main commands: query(), from and filter()

In ORM, queries are built around three fundamental commands: query(), from and filter(). Together, they define **what data** are retrieved (query), **where the data come from** (from) and **which records** are included in the result (filter).

3.1.1 The query() method

The query() method defines which columns are returned by an ORM query, and therefore determines the structure of the result set. It is the first clause users encounter when writing a query and answers the question "what information do I want to see?". The columns listed in query() can come from one or more tables, and they can also include expressions, aliases or functions.

The following example illustrates the core principle of ORM: the table in the database is represented by a Python object (TLS201_APPLN), and each column is accessed as an attribute of that object using dot notation (e.g. TLS201_APPLN.appln_id). In this way, ORM queries express database structure directly through objects and attributes, without referring explicitly to table or column names as strings. Remember that the attributes should be separated by commas.

```
db.query(
        TLS201_APPLN.appln_id,
        TLS201_APPLN.appln_auth,
        TLS201_APPLN.appln_filing_date
)
```

To view all the data in the table requires a slightly more advanced query.

```
cols = list(TLS201_APPLN.__table__.columns)
q = (db.query(*cols))
```

While this type of query can be useful for quick overviews or for checking the structure of a table, it is not suited to data analysis. Selecting all columns can slow down queries and produce unnecessarily large result sets. In practice, it is preferable to select only the required attributes in the db.query() call, as shown below.

Note: in Python, the asterisk (*) placed in front of a variable is the unpacking operator. When used with a list or tuple of columns (for example *cols), it expands that

collection into individual elements. In an ORM query, this allows a list of column objects to be passed to db.query() as if each column had been listed explicitly.

In query() you can use column aliases (or "**labels**") to improve the readability of the output without changing the underlying data. Labels are particularly useful in PATSTAT, where column names are often technical or abbreviated.

```
db.query(
        TLS201_APPLN.appln_auth.label("filing_authority"),
        TLS201_APPLN.appln_filing_date.label("filing_date")
)
```

Note: Labelling only some fields and not others in an ORM query may result in errors when the result is converted to a dataframe. It is best practice to label either all selected fields or none.

In relational databases, data are often structured in a way that allows the same value to appear multiple times in one column when it is associated with different values in another column. This is a natural consequence of normalisation and of one-to-many or many-to-many relationships.

For example, in PATSTAT the *docdb_family_id* identifies a patent family. A single family may contain several patent applications, each with its own application identifier (*appln_id*). As a result, the same *docdb_family_id* can appear in multiple rows of the applications table, once for each application belonging to that family. The distinct() keyword is used to **remove duplicate** rows from the result set.

```
db.query(
        TLS201_APPLN.docdb_family_id
)
.distinct()
```

Here, distinct() ensures that each DOCDB family identifier appears only once, even if it is associated with multiple different applications in the tls201_appln table. So a "SELECT COUNT(DISTINCT (docdb_family_id)) FROM tls201_appln" will return the number of unique DOCDB families in the PATSTAT database, while a "SELECT COUNT(DISTINCT (appln_id)) FROM tls201_appln" will return the number of unique applications.

3.1.2 The from command

The ORM equivalent of the SQL FROM clause involves importing the required ORM-mapped table objects in advance. Each database table is represented by a Python class, which must be imported before it can be used in a query. For example:

```
from epo.tipdata.patstat.database.models import TLS201_APPLN
```

Once imported, these classes define the dataset from which data are retrieved and can be accessed directly in ORM queries. The table itself is referenced using the class name (e.g. TLS201_APPLN), while individual columns are accessed as attributes of that class using dot notation (e.g. TLS201_APPLN.appln_id). These objects can also be assigned aliases.

```
from sqlalchemy.orm import aliased
from epo.tipdata.patstat.database.models import TLS201_APPLN
a = aliased(TLS201_APPLN)
q = db.query(
        a.appln_id,
        a.appln_auth
)
```

Assigning table aliases significantly improves readability and becomes essential when queries involve multiple tables or when tables share the same column names.

3.1.3 The filter() method

The filter() method refines ORM queries by returning only the records that meet your specified criteria. It serves as the primary mechanism for narrowing an analysis to a particular time period, country, applicant, role or other attribute.

In ORM, filtering occurs after the source tables are defined but before the final result set is generated. Conceptually, the ORM identifies the entire dataset from your referenced tables and then prunes it using the conditions in filter(). Because this filtering happens **before** the final selection, **aliases created using label() are unavailable for use within the filter() logic**.

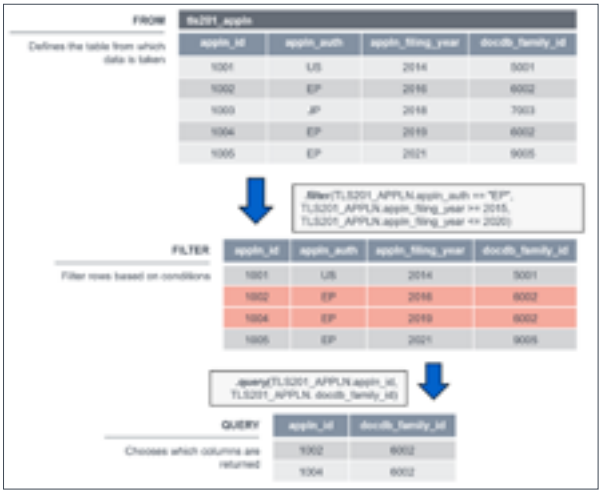
To apply multiple filters, simply separate conditions with commas for AND logic. For OR logic, use the | operator and ensure each condition is enclosed in parentheses.

```
db.query(TLS201_APPLN.docdb_family_id)
.filter(
        TLS201_APPLN.appln_auth == "EP",
        (TLS201_APPLN.appln_filing_year == 2015) |
        (TLS201_APPLN.appln_filing_year == 2020)
)
.distinct()
```

The filter() method supports a range of comparison operators.

Equality is checked using ==, while **inequality**, including comparisons such as !=, <, >, <=, and >=, can be applied to both numeric values and dates.

To check a column's value against a list of values, use the .in_() method. For text-based pattern matching, the like() operator supports wildcards, such as %.



```
db.query(
        TLS211_PAT_PUBLN.pat_publn_id,
        TLS211_PAT_PUBLN.publn_kind
)
.filter(
        TLS211_PAT_PUBLN.publn_date >= '2015-01-01',
        TLS211_PAT_PUBLN.publn_date <= '2020-12-31',
        TLS211_PAT_PUBLN.publn_auth.in_(["EP"]),
        TLS211_PAT_PUBLN.publn_kind.like("B%")
)
.distinct()
```

3.2 The join() method

To minimise redundancy and maintain data integrity, relational databases often distribute information across multiple tables. As a result, meaningful analyses typically require data from more than one table. The join() method is used to **combine rows from different tables** based on their logical relationships. As with the from command, you can use aliases to rename tables, ensuring that even complex queries remain readable.
In ORM, the join() method explicitly combines mapped table objects by defining the logical link between them, usually via primary foreign key relationships. **The join condition dictates exactly which columns must align** for rows to be combined. Once joined, the query can access attributes from any associated object.
As with filtering, aliases defined for output columns using label() cannot be referenced within the join condition as they are created only after the join is evaluated. To prevent ambiguity and ensure clarity, it is best practice to qualify column references with their respective table object names or aliases.

```
from epo.tipdata.patstat.database.models import TLS201_APPLN, TLS207_PERS_APPLN
db.query(
        TLS201_APPLN.appln_nr,
        TLS201_APPLN.appln_auth,
        TLS207_PERS_APPLN.applt_seq_nr,
        TLS207_PERS_APPLN.invt_seq_nr,
)
.join(
        TLS207_PERS_APPLN,
        TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id
)
```

Technically, the order in which tables are listed within a join condition does not impact the result. However, particularly in queries involving multiple joins, it is best practice to lead with the primary table you wish to analyse. This “top-down” approach makes the query’s logical structure much easier for others to follow.

3.2.1 Common types of joins

ORM supports various join types, each dictating how matching rows are selected. As a consequence, your choice of join type determines whether absent or incomplete relationships between tables appear in the results. In a highly normalised database such as PATSTAT, where data is distributed across many interrelated tables, selecting the appropriate join type is vital for ensuring your analysis accurately reflects the underlying information

Regardless of the join type used, every ORM join requires an explicit join condition. This condition serves as the logical bridge between tables, pinpointing the exact attributes that must align for rows to be combined.
Inner join: this is the default join type in ORM and is introduced using the join() method. It restricts the result set to rows that have a corresponding **match in both tables**. If no corresponding match is found, that row is excluded from the result set.

This type of join is typically used when your analysis requires data from both tables to be valid.
Left outer join: introduced using the outerjoin() method, this returns **all rows from the lefthand table** – the one defined first in your from clause – regardless of whether a matching record exists in the righthand table. If a match exists in the right-hand table, the data are merged; if not, the missing values are populated with NaN.
This join type is particularly useful when the absence of related information is analytically meaningful.

```
db.query(
        TLS201_APPLN.appln_id, TLS201_APPLN.appln_nr_epodoc,        TLS207_PERS_APPLN.applt_seq_nr,
TLS207_PERS_APPLN.invt_seq_nr
)
.outerjoin(
        TLS207_PERS_APPLN,
        TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id
)
```

Full outer join: implemented by adding the full=True parameter to a standard join command, this join type returns every record from both tables, merging them where a match exists. When a record in one table has no match in the other, NaN is used in place of the missing values.

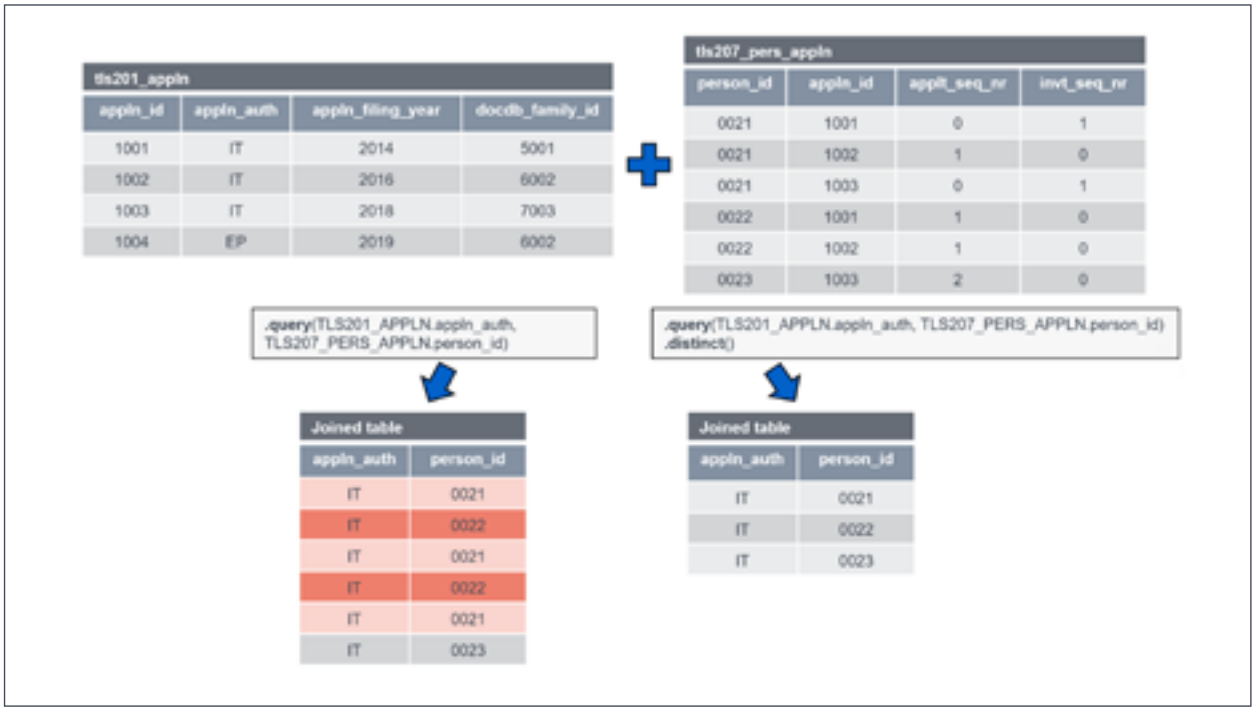
While useful for completeness checks or exploratory research, this join type can significantly inflate result sets and should be used with caution.

```
db.query(
        TLS201_APPLN.appln_id, TLS201_APPLN.appln_nr_epodoc,        TLS207_PERS_APPLN.applt_seq_nr, TLS207_
PERS_APPLN.invt_seq_nr
)
.join(
        TLS207_PERS_APPLN,
        TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id,
        full=True
)
```

3.2.2 Joins and row multiplication

A side effect of joins is the potential for row multiplication. When a single record in one table matches multiple records in another, the ORM generates an output row for every possible combination. While this accurately reflects the data's underlying relationships, it requires careful handling during analysis, especially for counts and aggregations.

In PATSTAT, this is a common occurrence; joining applications to entities like persons, classifications or citations typically expands the result set – a single application may be linked to several persons, multiple classification symbols or multiple citations. As a consequence, these queries often utilise DISTINCT, aggregation functions or targeted filters to manage duplication.



To ensure correct results, joins in PATSTAT queries should adhere to the key relationships outlined in the PATSTAT Data Catalog, available on the PATSTAT website.

3.3 Grouping, counting and aggregating rows

When working with relational databases, analysis often moves beyond retrieving individual records to focus on summarising data across meaningful categories. ORM supports this type of analysis through grouping and aggregation, which allow users to transform row-level data into higher-level indicators such as counts, totals or averages. Note that clauses in ORM queries must follow a strict, predefined order. This sequence is a fundamental requirement of ORM and must be followed to ensure the query is valid, regardless of the database system used. The specific order is detailed in Section **3.6** below.

3.3.1 The group_by() method

Analysing relational data often requires shifting from individual records to a higher level of abstraction. The group_by() method allows you to do this by grouping rows that share identical values in one or more columns. The query then returns a single row for each unique combination of values in those columns.
Conceptually, group_by() partitions the dataset before any summary statistics are calculated. For example, grouping patent applications by filing year (appln_filing_year) allows for temporal trend analysis rather than record-by-record inspection. An important rule applies: every non-aggregated element in the query() method must also be included in the group_by() clause to ensure logical consistency between grouped and non-grouped data.

```
from sqlalchemy import func
db.query(
        TLS201_APPLN.appln_auth,
        func.count(TLS201_APPLN.appln_id.distinct()).label("number_of_applications")
)

.group_by(TLS201_APPLN.appln_auth)
```

3.3.2 Aggregating values

Once rows have been grouped, ORM provides aggregation functions to calculate summary statistics for each group. These functions operate on the rows within each group and return a single value per group. The most commonly used aggregation function is count(), which returns the number of rows (or distinct values) in each group. Other commonly used functions include sum(), avg(), min() and max(), for calculating totals, averages and extreme values. These functions are called in the query() method. Therefore, as outlined in 3.2, aliases assigned to output columns using label() cannot be used in join conditions, since they are only defined after the join is evaluated. To avoid ambiguity and maintain clarity, column references should be explicitly qualified with their corresponding table name or aliases. Additionally, you can combine distinct() with these functions to aggregate only on unique values.

3.3.3 The having() method

After data have been grouped and aggregated, you can use the having() method to filter the results based on group-level criteria. While conceptually similar to filter(), having() operates at the group level rather than on individual rows. It is evaluated after aggregation, allowing you to apply conditions to the results of aggregation functions – for example, selecting only those groups with more than a given number of applications.

```
from sqlalchemy import func
db.query(
        TLS201_APPLN.appln_auth,
        TLS201_APPLN.appln_filing_year,
        func.count(TLS201_APPLN.docdb_family_id.distinct()).label("number_of_families")
)
.group_by(
        TLS201_APPLN.appln_auth,
        TLS201_APPLN.appln_filing_year)
.having(func.count(TLS201_APPLN.docdb_family_id.distinct()) > 100)
```

Note: When applied at the query level, distinct() removes duplicate **rows** from the final result set based on all selected attributes. Conversely, when used inside an aggregation function (e.g. count()), it ensures that the calculation only considers **unique values** for that specific attribute.
If a query already includes group_by(), adding distinct() at the query level is generally superfluous, as group_by() already ensures uniqueness across the grouping columns. While including distinct() won't cause errors or alter the results, it adds a redundant operation.

3.4 Other ORM query modifiers

3.4.1 The order_by method

The order_by() method dictates the sequence of rows in your final result set. Since **ORM queries do not guarantee any particular ordering of results** by default, this method is essential whenever a specific sequence is needed. You can sort rows in ascending order (using .asc(), the default) or descending order (using .desc()) across one or more columns or expressions.
When multiple criteria are provided, ORM applies them hierarchically, from left to right.
As the final clause in a query, order_by() manages the presentation of data without altering the underlying records or aggregation results.

```
from sqlalchemy import func
db.query(
        TLS201_APPLN.appln_filing_year,
        func.count(TLS201_APPLN.docdb_family_id.distinct()) .label("number_of_families")
)
.group_by(TLS201_APPLN.appln_filing_year)
.order_by(func.count(TLS201_APPLN.docdb_family_id.distinct()).desc())
```

Note: Column aliases created with label() are **not available** to the order_by() method. As a result, you must explicitly repeat the full sorting expression, even if you have already assigned it a label for the query output.

3.4.2 The limit() method

The limit() method restricts the total number of rows returned by an ORM query. Since it is applied at the final stage of query execution – after filtering, grouping and sorting – it does not alter the underlying calculations, but merely limits how many rows appear in the final output. This makes limit() especially useful for query debugging or exploratory tasks in large, normalised databases like PATSTAT, where queries might otherwise return millions of records. Additionally, it is commonly used to create "top-N" output. When paired with order_by(), it can isolate specific subsets like the "ten most cited patents" or "five most recent filings". Restricting output in this way also optimises memory usage and enhances system responsiveness.

```
from sqlalchemy import func
db.query(
        TLS201_APPLN.appln_auth,
        func.count(TLS201_APPLN.docdb_family_id.distinct()).label("family_count"))
.group_by(TLS201_APPLN.appln_auth)
.order_by(func.count(TLS201_APPLN.docdb_family_id.distinct()).desc())
.limit(10)
```

3.4.3 String functions

String functions enable the transformation and manipulation of text data. In SQLAlchemy's ORM, these functions allow you to standardise, combine or parse strings without modifying the underlying data. For example, it is common to include upper() and lower() in a filter() clause when performing case-insensitive comparisons in PATSTAT, to ensure that variations in capitalisation do not affect the results.

Additionally, the string concatenation operator + (or the concat(*argument 1, argument 2, ...*) method) can merge disparate fields, such as application authority, number and kind, into a single identifier. Other string functions, such as substring(*argument, starting_character, number_of_characters_to_extract*), allow for the extraction of specific segments, which is particularly useful, for instance, when working with classification symbols or country codes.

```
from sqlalchemy import func
from epo.tipdata.patstat.database.models import (TLS201_APPLN, TLS209_APPLN_IPC)
db.query(
        func.upper(TLS201_APPLN.appln_auth),
        (TLS201_APPLN.appln_auth + TLS201_APPLN.appln_nr + TLS201_APPLN.appln_kind)
        .label("application_number"),
        # in this case func.substring extract characters from 1 to 4
        func.substring(TLS209_APPLN_IPC.ipc_class_symbol, 1, 4) .label("ipc_section"),
)
.join(
        TLS209_APPLN_IPC,
        TLS201_APPLN.appln_id == TLS209_APPLN_IPC.appln_id
)
.distinct()
```

3.4.4 Datetime functions

Datetime functions make it possible to analyse, organise and compare temporal data effectively. They are indispensable when you need to break up dates or timestamps into their individual components, filter records by specific periods and calculate the duration between events.

The extract() function is commonly used to retrieve numerical components, such as year, month or day, from a date. For measuring intervals, date_diff() returns the span between two dates in a specified unit, such as days or years. Conversely, date_trunc() is used to round a date or timestamp down to a specified level of granularity, such as year or month.

```
from sqlalchemy import func
from sqlalchemy.sql import literal_column
db.query(
  func.extract(
    "year", TLS201_APPLN.appln_filing_date
  ).label("filing_year"),
  func.date_trunc(
    TLS201_APPLN.appln_filing_date,
    literal_column("YEAR")
  ).label("filing_year_truncated"),
  func.date_diff(
    TLS201_APPLN.earliest_publn_date,
    TLS201_APPLN.appln_filing_date,
    literal_column("DAY"),
  ).label("filing_to_publication_delay")
)
```

Note: Certain database systems, such as BigQuery, require specific function arguments (for example date parts like YEAR, MONTH or DAY) to be provided as SQL keywords rather than as string values. When using ORM, passing these arguments as Python strings may cause them to be handled as bound parameters, leading to execution errors. The literal_column() function allows such arguments to be injected directly into the generated SQL as literal tokens, allowing the database to interpret the function calls correctly.

3.4.5 Arithmetic functions

ORM allows you to perform numerical calculations directly within queries using arithmetic functions and operators. At the row level, **standard mathematical operators** such as addition (+), subtraction (-), multiplication (*) and division (/), can be used to calculate ratios, percentages, or other derived values.
For broader analysis, aggregate functions such as sum(), avg(), count(), min() and max() summarise data across multiple rows. These functions must be used together with group_by().
Finally, scalar functions like round(), which rounds the number of decimal places, abs(), which returns the absolute value, and ceil() and floor(), which round numbers up or down, provide more granular control.

```
db.query(
        func.count(TLS201_APPLN.appln_id.distinct())
        .label("total_applications"),
        (func.count(TLS201_APPLN.appln_id.distinct()) /        func.count(TLS201_APPLN.docdb_family_id.distinct()))
         .label("applications_per_family"),
         func.round(func.avg(TLS201_APPLN.docdb_family_size), 1)
        .label("average_family_size"),
        func.max(TLS201_APPLN.docdb_family_size)
        .label("largest_family_size"),
)
```

3.5 Order of ORM query commands

ORM queries are built by combining a set of logical operations that define how data are selected, combined, filtered, aggregated and presented. These operations are expressed as chained method calls and together describe the structure and behaviour of the query.
While these methods do not need to be written in a fixed order, the **logical dependencies between operations must be respected.** For example, tables must be joined before their attributes can be referenced, grouping must be defined before applying the having() method, and ordering should be specified before limiting results.

Each logical operation is applied once per query, with the exception of joins, which may be repeated to combine multiple tables. When multiple conditions are required, they are combined within a single method call using logical operators, and multiple attributes are specified together rather than through repeated operations.

4. General tips for writing a query

4.1 Planning and defining the analysis

Before writing any Python query, it is essential to clearly define the analytical objective. This includes specifying the information to be retrieved and determining the appropriate level of aggregation, such as whether the analysis should be conducted at the level of patent families, applications, publications or some other relevant unit. At this stage, it is also important to identify any filter conditions that will restrict the dataset to the relevant scope, for example by country, time period, applicant type or technical field. Particular attention should be paid to the choice of date variables (such as priority date, filing date, publication date or grant date) as this has a direct impact on the interpretation of results.

At the same time, you should verify that the required information is actually available in PATSTAT. This includes checking whether data coverage is sufficient for the countries, offices or time periods of interest.

Once the objective is clear, the required data elements must be identified. This involves selecting the appropriate tables and attributes needed to define the query criteria, to correctly join different tables and to produce the final output. Consulting the PATSTAT Data Catalog is strongly recommended, as it provides authoritative information on table structures and attribute definitions.

4.2 Designing and testing queries

Query construction should follow an incremental and iterative approach. It is advisable to start with a simple query and progressively add joins, filters and aggregations as needed. After each modification, check the results to ensure that they remain consistent with expectations and that there is no unintended duplication or data loss. Exploratory test queries play an important role in this validation process, and restricting output size using clauses such as .limit() makes it easier to inspect results.
Performance should also be considered throughout the data processing workflow in Python. Since operations on large datasets can be resource-intensive, reducing the volume of data as early as possible can significantly improve efficiency. For this reason, it is better to apply targeted filters and select only the data necessary.
Whenever possible, results should be cross-checked against external benchmarks. For example, Espacenet can be used to validate individual records via the APPLN_NR_EPODOC identifier (a concatenation of PUBLN_AUTH and PUBLN_NR). Ultimately, query formulation is an iterative cycle of execution, validation and refinement.

4.3 Writing robust and reusable Python queries

Efficient Python design avoids unnecessary repetition. Rather than writing separate queries for different values of year, office or classification, you should consolidate them into flexible, dynamic structures. Python provides several mechanisms to support this, including the in_() operator for multiple-value comparisons and the or_() operator combined with .like() and wildcard characters for pattern matching.
Clear documentation further improves robustness. Python **comments** – using # for concise, single-line notes, and triple quotes (“”” or ‘’’) for expansive, multi-line explanations – are vital both for ensuring complex logic remains understandable to all users and for disabling blocks of code during testing.

```
# This is a single line comment.
’’’
This allows for:
Multi-line comments.”””
```

As query complexity increases, **common table expressions** (CTEs) provide an effective way to improve structure and readability. Defined using the .subquery() command, a CTE acts as a temporary, named result set that can be referenced later in the query like a standard table. CTEs exist only for the duration of the query and do not create persistent database objects.
In practice, CTEs are ideal for breaking up complex logic into smaller, well-defined steps. By eliminating repeated subqueries, they make code easier to debug and maintain. They can include joins, filters and aggregations and can be referenced multiple times within the main query.

```
applicant_apps = (
        db.query(
                TLS201_APPLN.appln_id,
                TLS201_APPLN.appln_filing_year,
                TLS201_APPLN.docdb_family_id,
        )
        .filter(TLS201_APPLN.appln_filing_year.between(2015, 2020))
        .subquery()
)
q = (
        db.query(
                applicant_apps.c.appln_filing_year,
                func.count(applicant_apps.c.docdb_family_id.distinct())
                .label(“nb_docdb_families”),
        )
        .group_by(applicant_apps.c.appln_filing_year)
)
```

5. Getting started with Python queries for PATSTAT

NOTE: A companion Jupyter Notebook containing all the Python queries in this section is available in the "Getting started" folder within the default TIP training set.

NOTE: To run queries on TIP, you must initialise the PATSTAT client and create a database session within your Python environment (e.g. Jupyter) before executing any queries.

The following examples use a 'TEST' dataset designed for demonstration purposes. The dataset is selected when initialising the PATSTAT client (see the code snippet below).

To switch to the full production environment, update the initialisation parameter to 'PROD', i.e. PatstatClient(env='PROD')

```
#Import the PATSTAT client
from epo.tipdata.patstat import PatstatClient

# Initialise the PATSTAT client
patstat = PatstatClient(env='TEST')
db = patstat.orm()

# Import the libraries
from sqlalchemy.orm import aliased
from sqlalchemy import func
from sqlalchemy.sql import literal_column
import pandas as pd
```

NOTE: Python queries in TIP are case-sensitive for text data, including applicant names and IPC/CPC codes.

NOTE: All queries in this section use the limit() method. Applying limit(10) restricts the output to a maximum of 10 rows, significantly improving query efficiency. You can increase this number if more rows are required, or remove the limit()entirely to retrieve all results.
If you want to view only the first 10 rows without limiting the amount of *data retrieved*, you can instead use df.head(10) at the end of your query. However, keep in mind that this still processes the entire dataset, whereas limit(10) stops processing once the threshold is met.

Query 1: List all granted EP applications with filing year = 2010.

```
from sqlalchemy import func

q = (
  db.query(
    TLS201_APPLN.appln_id,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr,
      TLS201_APPLN.appln_kind
    ).label("number"),
    TLS201_APPLN.appln_filing_date
  )
  .filter(
    TLS201_APPLN.appln_filing_year == 2010,
    TLS201_APPLN.appln_auth == "EP",
    TLS201_APPLN.granted == "Y"
  )
.limit(10)
)

df1 = patstat.df(q)
df.1
```

Query 2: List all EP applications filed from 2010 to 2015 with CPC classification symbol = 'Y02E 10/7%' (wind energy).

NOTE: The CPC classification symbol must have 8 characters before the '/', so add spaces if needed.

```
from epo.tipdata.patstat.database.models import TLS201_APPLN, TLS224_APPLN_CPC
from sqlalchemy import func

q = (
  db.query(
    TLS201_APPLN.appln_id,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr,
      TLS201_APPLN.appln_kind,
    ).label("number"),
    TLS201_APPLN.appln_filing_date,
    TLS224_APPLN_CPC.cpc_class_symbol
  )
  .join(
    TLS224_APPLN_CPC,
    TLS201_APPLN.appln_id == TLS224_APPLN_CPC.appln_id
  )
  .filter(
    TLS201_APPLN.appln_filing_year.between(2010, 2015),
    TLS201_APPLN.appln_auth == "EP",
    TLS224_APPLN_CPC.cpc_class_symbol.like("Y02E  10/7%")
  )
  .order_by(TLS201_APPLN.appln_id)
  .limit(10)
)

df2 = patstat.df(q)
df2
```

Query 3: List all applications filed by 'NOKIA NETWORKS'. For greater reliability, we used the PATSTAT standardised name (psn_name) instead of person_name.

```
from epo.tipdata.patstat.database.models import TLS201_APPLN, TLS202_APPLN_TITLE, TLS207_PERS_APPLN,
TLS206_PERSON
from sqlalchemy import func, desc

q = (
  db.query(
    TLS201_APPLN.appln_id,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr,
     TLS201_APPLN.appln_kind,
    ).label("number"),
    TLS201_APPLN.appln_filing_date,
    TLS206_PERSON.psn_name,
    TLS202_APPLN_TITLE.appln_title
  )
  .join(
    TLS207_PERS_APPLN,
    TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id
  )
  .join(
    TLS206_PERSON,
    TLS207_PERS_APPLN.person_id == TLS206_PERSON.person_id
  )
  .join(
    TLS202_APPLN_TITLE,
    TLS202_APPLN_TITLE.appln_id == TLS201_APPLN.appln_id
  )
  .filter(
    TLS206_PERSON.psn_name == "NOKIA NETWORKS"
  )
  .order_by(desc(TLS201_APPLN.appln_filing_date))
  .limit(10)
)

df3 = patstat.df(q)
df3
```

Query 4: Generate a top-ten list of Chinese applicants based on the number of EP applications.

```
from epo.tipdata.patstat.database.models import TLS201_APPLN, TLS202_APPLN_TITLE, TLS207_PERS_APPLN,
TLS206_PERSON
from sqlalchemy import func, desc

q = (
  db.query(
    TLS206_PERSON.psn_name,
    TLS206_PERSON.person_ctry_code,
    func.count(TLS201_APPLN.appln_id)
      .label("applications_at_epo")
  )
  .join(
    TLS207_PERS_APPLN,
    TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id
  )
  .join(
    TLS206_PERSON,
    TLS207_PERS_APPLN.person_id == TLS206_PERSON.person_id
  )
  .filter(
    TLS206_PERSON.person_ctry_code == "CN",
    TLS207_PERS_APPLN.applt_seq_nr > 0,
    TLS207_PERS_APPLN.invt_seq_nr == 0,
    TLS201_APPLN.appln_auth == "EP"
  )
  .group_by(
    TLS206_PERSON.psn_name,
    TLS206_PERSON.person_ctry_code
  )
  .order_by(desc("applications_at_epo"))
  .limit(10)
)

df4 = patstat.df(q)
df4
```

Query 5: Generate a list of granted European patent applications filed by Chinese applicants for which a renewal fee was paid in Belgium.
The 'appln_auth == 'EP'' in the .filter() clause is unnecessary since 'event_code == 'PGFP'' already indicates a granted EP patent. Application IDs are included for statistical purposes.
Note: Paying a renewal fee does not guarantee the patent is currently valid.

```
from epo.tipdata.patstat.database.models import TLS201_APPLN, TLS207_PERS_APPLN, TLS206_PERSON, TLS231_
INPADOC_LEGAL_EVENT
from sqlalchemy import func, desc

q = (
  db.query(
    TLS206_PERSON.psn_name,
    TLS206_PERSON.person_ctry_code,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr,
      TLS201_APPLN.appln_kind,
    ).label("number"),
    TLS201_APPLN.appln_filing_date,
    TLS231_INPADOC_LEGAL_EVENT.fee_country,
    TLS231_INPADOC_LEGAL_EVENT.fee_payment_date,
    TLS231_INPADOC_LEGAL_EVENT.fee_renewal_year,
    TLS202_APPLN_TITLE.appln_title
  )
  .distinct()
  .join(
    TLS207_PERS_APPLN,
    TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id
  )
  .join(
    TLS206_PERSON,
    TLS207_PERS_APPLN.person_id == TLS206_PERSON.person_id
  )
  .join(
    TLS231_INPADOC_LEGAL_EVENT,
    TLS201_APPLN.appln_id == TLS231_INPADOC_LEGAL_EVENT.appln_id
  )
  .join(
    TLS202_APPLN_TITLE,
    TLS201_APPLN.appln_id == TLS202_APPLN_TITLE.appln_id
  )
  .filter(
    TLS206_PERSON.person_ctry_code == "CN",
    TLS207_PERS_APPLN.applt_seq_nr > 0,
    TLS201_APPLN.appln_auth == "EP",
    TLS231_INPADOC_LEGAL_EVENT.event_code == "PGFP",
    TLS231_INPADOC_LEGAL_EVENT.fee_country == "BE"
  )
  .order_by(desc(TLS231_INPADOC_LEGAL_EVENT.fee_payment_date))
  .limit(10)
)
df5 = patstat.df(q)
df5
```

Query 6: Find the ten most cited applications filed in Great Britain.
This query uses the attribute nb_citing_docdb_fam, which indicates how many unique DOCDB families have cited the application or any of its family members. Often, citation frequency at the family level is more meaningful than counting citations for individual publications.

```
from epo.tipdata.patstat.database.models import TLS201_APPLN
from sqlalchemy import func, desc

q = (
  db.query(
    TLS201_APPLN.nb_citing_docdb_fam,
    TLS201_APPLN.appln_id,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr,
      TLS201_APPLN.appln_kind,
    ).label("number"),
    TLS201_APPLN.appln_filing_date
  )
  .filter(
    TLS201_APPLN.appln_auth == "GB"
  )
  .order_by(desc(TLS201_APPLN.nb_citing_docdb_fam))
  .limit(10)
)

df6 = patstat.df(q)
df6
```

Query 7: Find the most active Italian applicants. To select applicants and exclude only inventors, use *applt_seq_nr* > 0 AND invt_seq_nr = 0. Applicants are filtered by country of residence, which may skew results if multinational corporations file centrally.

```
from epo.tipdata.patstat.database.models import TLS206_PERSON, TLS207_PERS_APPLN
from sqlalchemy import func, desc

q = (
  db.query(
    TLS206_PERSON.psn_name,
    TLS206_PERSON.person_ctry_code,
    func.count(TLS201_APPLN.appln_id)
      .label("number_of_applications")
  )
  .join(
    TLS207_PERS_APPLN,
    TLS206_PERSON.person_id == TLS207_PERS_APPLN.person_id
  )

  .join(
    TLS201_APPLN,
    TLS207_PERS_APPLN.appln_id == TLS201_APPLN.appln_id
  )
  .filter(
    TLS206_PERSON.person_ctry_code == "IT",
    TLS207_PERS_APPLN.applt_seq_nr > 0,
    TLS207_PERS_APPLN.invt_seq_nr == 0
  )
  .group_by(
    TLS206_PERSON.psn_name,
    TLS206_PERSON.person_ctry_code
  )
  .order_by(desc("number_of_applications"))
  .limit(10)
)

df7 = patstat.df(q)
df7
```

Query 8.1: List EP patents having a Belgian applicant and non-Belgian co-applicants.

NOTE: In Python, you might want to create a table alias before referencing it in the query.

```
# Creation of alias

from epo.tipdata.patstat.database.models import TLS206_PERSON, TLS207_PERS_APPLN
from sqlalchemy.orm import aliased

P1 = TLS206_PERSON
PA1 = TLS207_PERS_APPLN

P2 = aliased(TLS206_PERSON)      # alias of tls206_person
PA2 = aliased(TLS207_PERS_APPLN)  # alias of tls207_pers_appln

# Query

from sqlalchemy import func

q = (
  db.query(
    P1.psn_name,
    P1.person_ctry_code,
    P2.psn_name,
    P2.person_ctry_code,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr
    ).label("number")
  )
  .distinct()
  .join(PA1, P1.person_id == PA1.person_id)
  .join(PA2, PA1.appln_id == PA2.appln_id)
  .join(P2, PA2.person_id == P2.person_id)
  .join(TLS201_APPLN, PA1.appln_id == TLS201_APPLN.appln_id)
  .filter(
    P1.person_ctry_code == "BE",
    PA1.applt_seq_nr > 0,
    PA2.applt_seq_nr > 0,
    P1.person_ctry_code != P2.person_ctry_code,
    TLS201_APPLN.appln_auth == "EP"
  )
  .order_by(P1.psn_name)
  .limit(10)

)

df8_1 = patstat.df(q)
df8_1
```

Query 8.2: This version includes the *international* co-applicants. The Belgian and non-Belgian co-applicant pairs are organised based on how many joint applications they have submitted.

```
from sqlalchemy import func, desc, literal_column
from sqlalchemy.orm import aliased

P1 = TLS206_PERSON
PA1 = TLS207_PERS_APPLN

P2 = aliased(TLS206_PERSON)
PA2 = aliased(TLS207_PERS_APPLN)

q = (
  db.query(
    func.count().label("number_of_common_applications"),
    P1.psn_name.label("name1"),
    P1.person_ctry_code.label("cc1"),
    literal_column("'co-applicant with'").label("relation"),
    P2.psn_name.label("name2"),
    P2.person_ctry_code.label("cc2")
  )
  .join(PA1, P1.person_id == PA1.person_id)
  .join(PA2, PA1.appln_id == PA2.appln_id)
  .join(P2, PA2.person_id == P2.person_id)
  .filter(
    P1.person_ctry_code == "BE",
    PA1.applt_seq_nr > 0,
    PA2.applt_seq_nr > 0,
    P1.person_ctry_code = P2.person_ctry_code
  )
  .group_by(
    P1.psn_name,
    P1.person_ctry_code,
    P2.psn_name,
    P2.person_ctry_code
  )
  .order_by(
    desc("number_of_common_applications"),
    P1.psn_name.asc(),
    P2.psn_name.asc()
  )
  .limit(10)
)

df8_2 = patstat.df(q)
df8_2
```

Query 9: Find the applications that were filed in Italy and where the inventor was also the applicant.

The condition "(*applt_seq_nr* > 0) AND (*invt_seq_nr* > 0)" returns both applicants and inventors.

```
from epo.tipdata.patstat.database.models import TLS206_PERSON, TLS201_APPLN, TLS207_PERS_APPLN
from sqlalchemy import func

q = (
  db.query(
    TLS206_PERSON.psn_name,
    TLS201_APPLN.appln_auth,
    TLS201_APPLN.appln_nr,
    TLS201_APPLN.appln_kind,
    TLS201_APPLN.appln_filing_date
  )
  .join(
    TLS207_PERS_APPLN,
    TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id
  )
  .join(
    TLS206_PERSON,
    TLS207_PERS_APPLN.person_id == TLS206_PERSON.person_id
  )
  .filter(
    TLS201_APPLN.appln_auth == "IT",
    TLS207_PERS_APPLN.applt_seq_nr > 0,
    TLS207_PERS_APPLN.invt_seq_nr > 0
  )
  .limit(10)
)

df9 = patstat.df(q)
df9
```

Query 10: Find the first filings of the VESTAS company in 2020.

Company name variations are handled using the '%' wildcard character.

```
from epo.tipdata.patstat.database.models import TLS206_PERSON, TLS201_APPLN, TLS207_PERS_APPLN, TLS206_
PERSON
from sqlalchemy import func

q = (
  db.query(
    TLS206_PERSON.psn_name,
    TLS201_APPLN.appln_id,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr
    ).label("number")
  )
  .join(
    TLS207_PERS_APPLN,
    TLS201_APPLN.appln_id == TLS207_PERS_APPLN.appln_id
  )
  .join(
    TLS206_PERSON,
    TLS207_PERS_APPLN.person_id == TLS206_PERSON.person_id
  )
  .filter(
    TLS201_APPLN.appln_filing_date >= "2020-01-01",
    TLS201_APPLN.appln_filing_date <= "2020-12-31",
    TLS201_APPLN.appln_id == TLS201_APPLN.earliest_filing_id,
    TLS207_PERS_APPLN.applt_seq_nr > 0,
    TLS206_PERSON.psn_name.like("VESTAS%")
  )
  .limit(10)
)

df10 = patstat.df(q)
df10
```

Query 11: Retrieve all A1 publications published by the USPTO in Q1/2009.

```
from epo.tipdata.patstat.database.models import TLS211_PAT_PUBLN
from sqlalchemy import func, literal_column

q = (
  db.query(
    func.concat(
      TLS211_PAT_PUBLN.publn_auth,
      literal_column("' '"),
      TLS211_PAT_PUBLN.publn_nr,
      literal_column("' '"),
      TLS211_PAT_PUBLN.publn_kind
    ).label("PublNr"),
    TLS211_PAT_PUBLN.publn_date
  )
  .filter(
    TLS211_PAT_PUBLN.publn_auth == "US",
    TLS211_PAT_PUBLN.publn_kind == "A1",
    TLS211_PAT_PUBLN.publn_date >= "2009-01-01",
    TLS211_PAT_PUBLN.publn_date <= "2009-03-31"
  )
  .order_by(TLS211_PAT_PUBLN.publn_date)
)

  .limit(10)
)

df11 = patstat.df(q)
df11
```

Query 12.1: Retrieve all applications with both 'C01B' and 'H01M' IPC classification symbols

```
from epo.tipdata.patstat.database.models import TLS209_APPLN_IPC, TLS201_APPLN
from sqlalchemy import exists, desc


# subquery EXISTS per C01B
exists_c01b = (
  db.query(TLS209_APPLN_IPC.appln_id)
  .filter(
    TLS209_APPLN_IPC.appln_id == TLS201_APPLN.appln_id,
    TLS209_APPLN_IPC.ipc_class_symbol.like("C01B%")
  )
  .exists()
)

# subquery EXISTS per H01M
exists_h01m = (
  db.query(TLS209_APPLN_IPC.appln_id)
  .filter(
    TLS209_APPLN_IPC.appln_id == TLS201_APPLN.appln_id,
    TLS209_APPLN_IPC.ipc_class_symbol.like("H01M%")
  )
  .exists()
)

q = (
  db.query(
    TLS201_APPLN.appln_id,
    TLS201_APPLN.appln_auth,
    TLS201_APPLN.appln_nr,
    TLS201_APPLN.appln_kind,
    TLS201_APPLN.appln_filing_date
  )
  .filter(
    exists_c01b,
    exists_h01m
  )
  .order_by(desc(TLS201_APPLN.appln_filing_date))
  .limit(10)
)

df12_1 = patstat.df(q)
df12_1
```

If you prefer not to filter using the broader IPC subclass 'H01M', and instead want to narrow it down specifically to 'H01M 4/583', change the condition from TLS209_APPLN_IPC.ipc_class_symbol.like("H01M%") to TLS209_APPLN_IPC.ipc_class_symbol.like("H01M<space> <space> <space> 4/583").

Note:

- Multiple spaces (<space>) should be replaced by a single space character. Because PDF formatting limitations mean that it is not always possible to reproduce consistent spacing, this notation is used throughout the document.
- Observe the three spaces in the subclass 'H01M<s pace><space><space>4/583'. The IPC (or CPC) main group number (in this example, number 4) must always occupy four positions and be right-aligned, with additional spaces added as necessary to meet this standard. This formatting aligns fully with WIPO Standard ST.8.
- To retrieve applications with an exact IPC subclass match, use '==' instead of .like(). When using '==', remove the % symbol, as this serves as a wildcard only when combined with the .like() operator.

Query 12.2: To return the number of applications by country that are categorised under both IPC symbols 'C01B' and 'H01M', you will need to make a small adjustment to the query:

```
from sqlalchemy import func, desc
from epo.tipdata.patstat.database.models import TLS209_APPLN_IPC, TLS201_APPLN

# EXISTS per C01B
exists_c01b = (
  db.query(TLS209_APPLN_IPC.appln_id)
  .filter(
    TLS209_APPLN_IPC.appln_id == TLS201_APPLN.appln_id,
    TLS209_APPLN_IPC.ipc_class_symbol.like("C01B%")
  )
  .exists()
)

# EXISTS per H01M
exists_h01m = (
  db.query(TLS209_APPLN_IPC.appln_id)
  .filter(
    TLS209_APPLN_IPC.appln_id == TLS201_APPLN.appln_id,
    TLS209_APPLN_IPC.ipc_class_symbol.like("H01M%")
  )
  .exists()
)

q = (
  db.query(
    TLS201_APPLN.appln_auth,
    func.count(TLS201_APPLN.appln_id)
      .label("number_of_applications")
  )
  .filter(
    exists_c01b,
    exists_h01m
  )
  .group_by(
    TLS201_APPLN.appln_auth
  )
  .order_by(
    desc("number_of_applications")
  )
  .limit(10)
)

df12_2 = patstat.df(q)
df12_2
```

Query 12.3: Retrieving all applications that include a particular IPC class or group, among other criteria, is much more straightforward and efficient.

```
from sqlalchemy import func
from epo.tipdata.patstat.database.models import TLS209_APPLN_IPC, TLS201_APPLN

q = (
  db.query(
    TLS201_APPLN.appln_id,
    func.concat(
      TLS201_APPLN.appln_auth,
      TLS201_APPLN.appln_nr
    ).label("application_number"),
    TLS209_APPLN_IPC.ipc_class_symbol
  )
  .join(
    TLS209_APPLN_IPC,
    TLS201_APPLN.appln_id == TLS209_APPLN_IPC.appln_id
  )
  .filter(
    TLS209_APPLN_IPC.ipc_class_symbol.like("B60K%")
  )
  .order_by(
    TLS201_APPLN.appln_id
  )
  .limit(10)
)

df12_3 = patstat.df(q)
df12_3
```

Query 13: Find which office published the most applications (filed in 2009) within 15 months. Publication usually occurs after 18 months, but these applications are published much earlier:

```
from sqlalchemy import func, desc, literal_column
from epo.tipdata.patstat.database.models import TLS201_APPLN
q = (
  db.query(
    TLS201_APPLN.appln_auth,
    func.count(TLS201_APPLN.appln_id).label("number")
  )
  .filter(
    TLS201_APPLN.appln_filing_year == 2009,
    func.date_add(
      TLS201_APPLN.appln_filing_date,
      literal_column("INTERVAL 15 MONTH")
    ) >= TLS201_APPLN.earliest_publn_date
  )
  .group_by(
    TLS201_APPLN.appln_auth
  )
  .order_by(
    desc("number")
  )
  .limit(10)
)

df13 = patstat.df(q)
df13
```

Query 14: Identify "VESTAS WIND SYSTEMS A/S" inventors and the technical fields of the patents they have filed since 2000.

The earliest filing date (refer to the Data Catalog for a detailed definition) is used, as this date is closest to the actual date of invention:

```
from sqlalchemy import func, literal_column
from sqlalchemy.orm import aliased
from epo.tipdata.patstat.database.models import TLS201_APPLN, TLS207_PERS_APPLN, TLS206_PERSON

# Alias for the subquery (same tables, NOT new tables)
A2  = aliased(TLS201_APPLN)
PA2 = aliased(TLS207_PERS_APPLN)
P2  = aliased(TLS206_PERSON)

# Subquery: appln_id with applicant "VESTAS%" and  earliest_filing_year >= 2000 ( < 9999)
subq_appln = (
  db.query(A2.appln_id)
  .join(PA2, A2.appln_id == PA2.appln_id)
  .join(P2,  PA2.person_id == P2.person_id)
  .filter(
    P2.psn_name.like("VESTAS%"),
    PA2.applt_seq_nr > 0,            A2.earliest_filing_year >= 2000,
    A2.earliest_filing_year < 9999
  )
)

# Main query
q = (
  db.query(
    TLS206_PERSON.psn_name.label("inventors_at_vestas"),
    TLS206_PERSON.person_ctry_code,
    func.string_agg(
      TLS209_APPLN_IPC.ipc_class_symbol,
      literal_column("', '")
    ).label("ipc_list")
  )
  .join(TLS207_PERS_APPLN, TLS206_PERSON.person_id == TLS207_PERS_APPLN.person_id)
  .join(TLS209_APPLN_IPC,  TLS207_PERS_APPLN.appln_id == TLS209_APPLN_IPC.appln_id)
  .filter(
    TLS207_PERS_APPLN.invt_seq_nr > 0,              TLS207_PERS_APPLN.appln_id.in_(subq_appln)      (subquery)
  )
  .group_by(
    TLS206_PERSON.psn_name,
    TLS206_PERSON.person_ctry_code
  )
  .limit(10)
)

df14 = patstat.df(q)
df14
```

Query 15: Retrieve all applications of the largest families in the dataset.

```
from sqlalchemy import desc
from epo.tipdata.patstat.database.models import TLS201_APPLN
q = (
  db.query(
    TLS201_APPLN.docdb_family_id,
    TLS201_APPLN.docdb_family_size,
    TLS201_APPLN.appln_id,
    TLS201_APPLN.appln_auth,
    TLS201_APPLN.appln_nr,
    TLS201_APPLN.appln_kind,
    TLS201_APPLN.appln_filing_date
  )
  .order_by(
    desc(TLS201_APPLN.docdb_family_size)
  )
  .limit(1000)
)

df15 = patstat.df(q)
df15.head(10)
```

Annex 1 | PATSAT FAQ

A1.1 What is the data coverage of PATSTAT Global?

PATSTAT Global uses DOCDB, the EPO's primary database for global patent information. Applications or publications not in DOCDB are excluded from PATSTAT. The only exceptions are replenished applications – artificial entries created to account for missing records. While DOCDB updates continuously, PATSTAT provides a snapshot of the data from three months prior to each edition. For more details, visit the PATSTAT coverage page.

A1.2 What are artificial applications/publications?

Application replenishment is the process of generating artificial applications (or publications) to account for un-linkable or unknown applications (publications). This includes instances where cited applications or publications lack a reference in DOCDB, such as Non-Patent Literature (NPL) citations. You can identify artificial applications by their ID: they always have *appln_id* ≥ 900,000,000. For further information, see the "Application Replenishment" section in the PATSTAT Data Catalog.

A1.3 Do the IDs in PATSTAT change from one edition to the next?

PATSTAT IDs are used for technical purposes only and have no inherent business meaning. While certain identifiers, such as *appln_id, pat_publn_id, person_id,* remain stable across editions, others may change. For further information, see the Surrogate Database Keys section in the Data Catalog.

A1.4 What does date '9999-12-31' signify?

Unknown publication dates are represented by the placeholder date '9999-12-31'. When filtering for records after a specific date, ensure you explicitly exclude this dummy date. The following example demonstrates how to do this when retrieving Danish publications from 2005 onwards.

```
from epo.tipdata.patstat.database.models import TLS211_PAT_PUBLN
cols = list(TLS211_PAT_PUBLN.__table__.columns)
q = (
  db.query(*cols)
  .filter(
    TLS211_PAT_PUBLN.publn_auth == "DK",
    TLS211_PAT_PUBLN.publn_date >= "2005-01-01",
    TLS211_PAT_PUBLN.publn_date < "9999-12-31"
  )
  .order_by(TLS211_PAT_PUBLN.publn_date.asc())
  .limit(1000)
)

dfA1_4 = patstat.df(q)
dfA1_4.head(10)
```

A1.5 There are so many different date fields. Which should I use?

The date field you use depends on what you're analysing, but the following general rules apply.

- For invention analysis, use the earliest filing date of the family.
- For application filings, use the date of filing.
- The date of the first publication is the date that the invention becomes prior art.

Note: None of these dates are available before the first publication.

A1.6 Should I count publications, applications or families?

This depends on the nature of the analysis.
Note: Counting documents (such as publications) differs from counting applications (filings in various offices), which is in turn distinct from counting inventions (families, defined as groups of identical or closely related applications).

A1.7 How can I retrieve applicant information?

- The *tls206_person* table lists applicants and inventors, including individuals, corporations and organisations.
- Link person names in *tls206_person* to applications in *tls201_appln* via *tls207_pers_appln*.
- Use tls227_pers_publn to join publication data with person tables for tracking applicant and inventor changes.
- Select applicants using *applt_seq_nr* > 0 and inventors with *invt_seq_nr* > 0. To find persons who are both, use (*applt_seq_nr* > 0) AND (*invt_seq_nr* > 0).
- This condition retrieves people who are listed as both applicant and inventor.
- To retrieve applicants, but exclude applicants that are also inventors, use (applt_seq_nr > 0) AND (invt_seq_nr = 0).

A1.8 How should I handle variations in applicant and inventor names?

PATSTAT contains data gathered from numerous national sources over an extended period. Due to the absence of an internationally agreed-upon unique identifier for inventors or applicants, the same company, individual or organisation can appear under various different names.

To resolve this, you should use name harmonisation. PATSTAT provides several harmonised naming options, including DOCDB standardised name, PATSTAT standardised name, and OECD HAN.

A1.9 How can I identify PCT applications? How can I tell which office was the receiving Office?

Filings

- The table *tls201_appln* contains records of international application filings, distinguished by *appln_kind* = 'W'.
- The attribute receiving_office specifies the receiving Office for each PCT application.
- The attribute appln_auth identifies the responsible authority – for all international applications, this is WO (WIPO).
- *internat_appln_id* > 0 provides a link between an international (PCT) application and subsequent later national/regional phases.
- In *tls201_appln*, the *int_phase* attribute (Y/N) indicates if an application is, or was, in the international phase.

Publications overview

- Table *tls211_pat_publn* lists WIPO international application publications (identified by *publn_auth* = 'WO').
- The *publn_nr* field contains the WO number in DOCDB format.

A1.10 How can I identify European patents that are in the national phase?

The republication of European patents at the national phase is complex. However, when a national office and the EPO exchange data in a structured and consistent way, these publications are integrated into simple DOCDB families and made available in both DOCDB and PATSTAT. You can track these by querying the INPADOC legal events table (*tls231_inpadoc_legal_event*) for the PGFP code. For detailed information, include the *fee_country*, *fee_payment_date*, and *fee_renewal_year* fields. For additional resources, see the Sources of information on national phase entries of EP patents.

A1.11 Why does some data appear to be missing, and how should I handle this?

Because PATSTAT Global aggregates data from various national offices, the quality and completeness of records can vary. For instance, many JP, CN and KR applications are missing country data for applicants or inventors.

Check for data completeness before analysis, and consider using related patent family records (such as PCT applications) to fill in the missing information.

Annex 2 | FAQ related to patent information

A2.1 What is patent information?

The **patent information tour** provides a brief guide to patent information and its uses.

A2.2 What are patent families? What are the differences between a simple (DOCDB) family and an extended (INPADOC) family?

- A patent family is a group of patent applications that cover the same or similar technical subject-matter. These applications are connected by priority claims.
- Patent families at the EPO fall into two categories:
  - DOCDB simple patent family: a collection of patent documents covering a single invention. The technical content covered by the applications is considered to be identical. All members of a simple patent family share exactly the same priorities.
  - INPADOC extended patent family: a collection of patent documents covering a technology. The technical content covered by the applications is similar, but not necessarily identical. Members of an extended patent family share at least one priority with at least one other member, either directly or indirectly.

A2.3 What are IPC and CPC?

- IPC (International Patent Classification): a global system used by patent offices to categorise patents. There are about 70 000 IPC codes covering different technical areas. More information on IPC is available on the WIPO website.
- CPC (Cooperative Patent Classification): extends the IPC and is jointly managed by the EPO and the US Patent and Trademark Office. It includes section Y for new technological developments, and offers greater detail through its classification hierarchy. Find out more on the CPC website, or explore the classification scheme in Espacenet.

A2.4 Kind codes, publication codes and legal event codes vary by patent office. Where can I find an overview?

The EPO publishes weekly updates to legal event codes and associated reference tables.

A2.5 Do I need to account for national differences in patent data?

Every country and patent organisation has its own unique characteristics that evolve over time. Local legislation applicant filing practices can sometimes impact statistics and data trends in unexpected ways. It is therefore important to consider the following, among other things, when working with patent data:

- unity of invention
- dual filings (utility model and patent)
- technical relations
- republications of granted regional patents (EP)
- deferred examination
- professors' privilege
- different data coverage and completeness of data delivered to the EPO

Example 1
Prior to around 2010, patent applications in Japan typically contained fewer claims than those filed elsewhere. On average, an invention submitted as a single application at the USPTO would be divided into three separate filings when submitted in Japan. South Korea exhibits filing trends similar to Japan in this regard.

Example 2
Following legislative changes at the USPTO, there was a sharp increase in the publication of applications (kind code A) after 2000.

Europäisches Patentamt
European Patent Office
Office européen des brevets

Follow us

- Visit **epo.org**
- Subscribe to our newsletter at **epo.org/newsletter**
- Listen to our podcast at **epo.org/podcast**

Published and edited by
European Patent Office
Munich
Germany

Design
European Patent Office