PostgreSQL Berlin Meetup

04 February, 2025

zalando

Agenda

200 participants

Two rooms

Six talks

Auditorium

- **19:15 - 19:45** - *Bruce Momjian:* Database in the AI Trenches
- **19:45 - 20:10** Networking, Q&A, and pizza
- **20:10 - 20:25** *Oleksandr Schulgin:* SQL: The Good, The Bad and The Ugly, Lightning talk
- **20:30** - **21:00** *Priyanka Chatterjee:* Your Postgres DB performance Radar : pgBadger
- **21:00 - 22:00** Networking, Q&A, and pizza

Festival

- **19:15 - 19:45** - *Kaarel Moppel:* Spot VMs and Postgres
- **19:45 - 20:10** Networking, Q&A, and pizza
- **20:10 - 20:25** *Felix Kunde:* Doing PostgreSQL as a service so nobody cares about Postgres. Oh no ? Lightning talk.
- **20:30** - **21:00** *Alexey Kondratov:* Serveless PostgreSQL: the journey from ~1s startup time to 10s and back
- **21:00 - 22:00** Networking, Q&A, and pizza

Wi-Fi

Simply connect to Zalando_Free

Future meet-ups

Talk to me if you want to present

zalando

Let’s go!

PostgreSQL

vs.

Postgres

PostgreSQL
vs.
Postgres

Can you name a problem that one of them has, but the other doesn’t?

PostgreSQL
vs.
POSTGRES

Can you name a problem that one of them has, but the other doesn’t?

Maybe like this?

Structured (English) Query Language

The Good, The Bad, and The Ugly.

SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent data base user.

…

Examples of such users are accountants, engineers, architects, and urban planners. It is for this class of users that SEQUEL is intended.

— SEQUEL: A Structured English Query Language

SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent data base user.

…

Examples of such users are accountants, engineers, architects, and urban planners. It is for this class of users that SEQUEL is intended.

— SEQUEL: A Structured English Query Language

Chamberlin, Donald D; Boyce, Raymond F (1974).
Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control.

🎉

```
postgres=# SELECT now() - '1974-07-01';
          ?column?
----------------------------
 18481 days 15:02:38.757149
(1 row)
```

🎉

```
postgres=# SELECT age(now(), '1974-07-01');
                  age
----------------------------------------
 50 years 7 mons 3 days 15:02:32.480721
(1 row)
```

In 1997, the Gartner Group reported that 80% of the world's business ran on [redacted] with over 200 billion lines of code and 5 billion lines more being written annually.

...

Testimony before the House of Representatives in 2016 indicated that [redacted] is still in use by many federal agencies. Reuters reported in 2017 that 43% of banking systems still used [redacted] with over 220 billion lines of [redacted] code in use.

...

During the COVID-19 pandemic and the ensuing surge of unemployment, several US states reported a shortage of skilled [redacted] programmers to support the legacy systems used for unemployment benefit management.

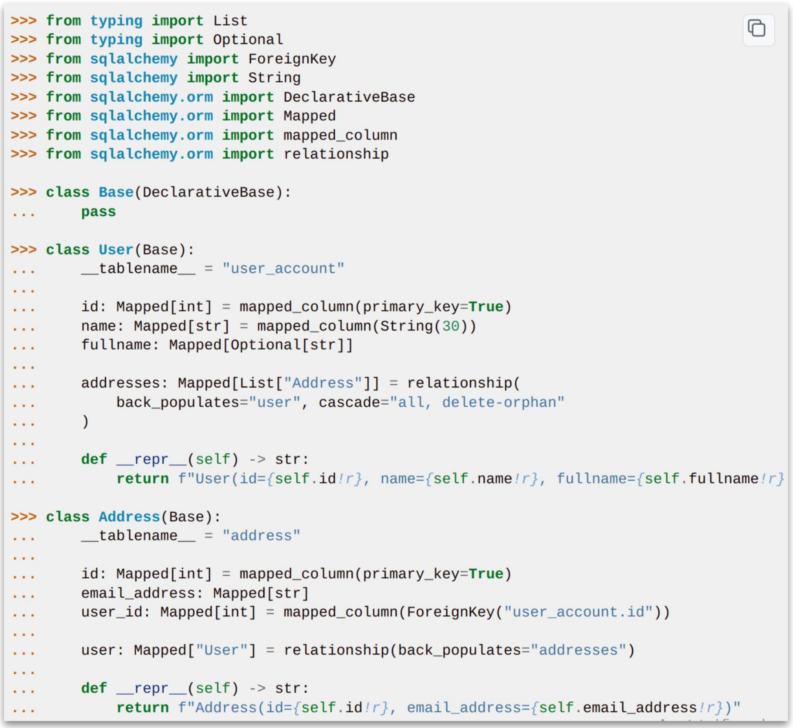
In 1997, the Gartner Group reported that 80% of the world's business ran on COBOL with over 200 billion lines of code and 5 billion lines more being written annually.

…

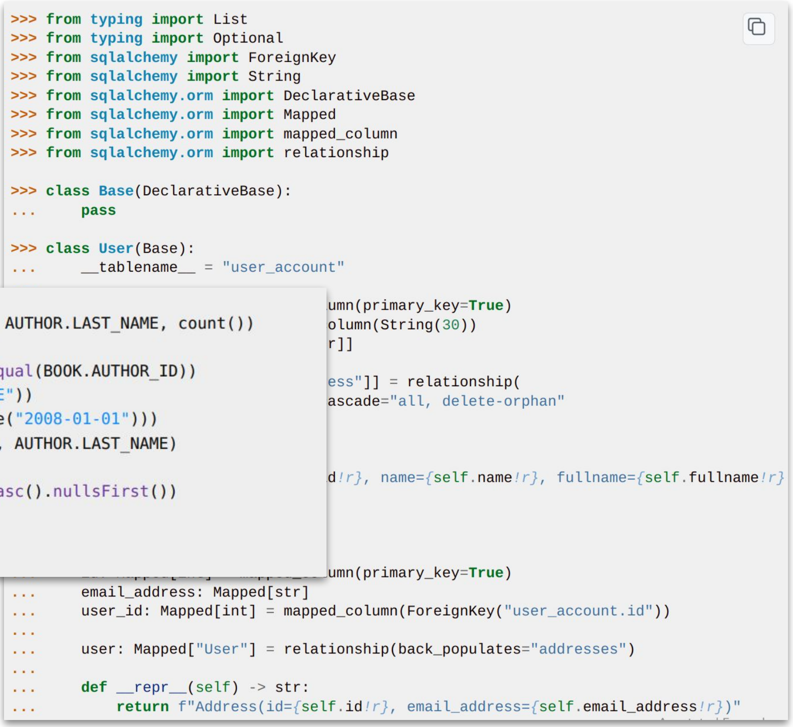
Testimony before the House of Representatives in 2016 indicated that COBOL is still in use by many federal agencies. Reuters reported in 2017 that 43% of banking systems still used COBOL with over 220 billion lines of COBOL code in use.

…

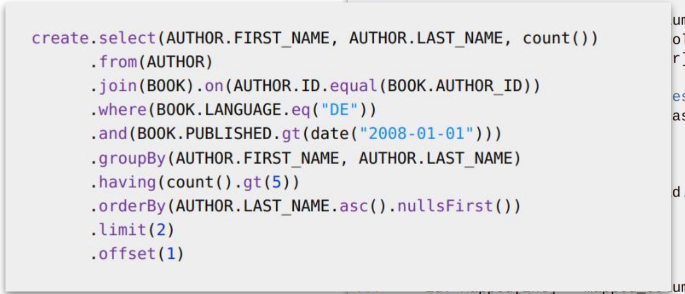
During the COVID-19 pandemic and the ensuing surge of unemployment, several US states reported a shortage of skilled COBOL programmers to support the legacy systems used for unemployment benefit management.



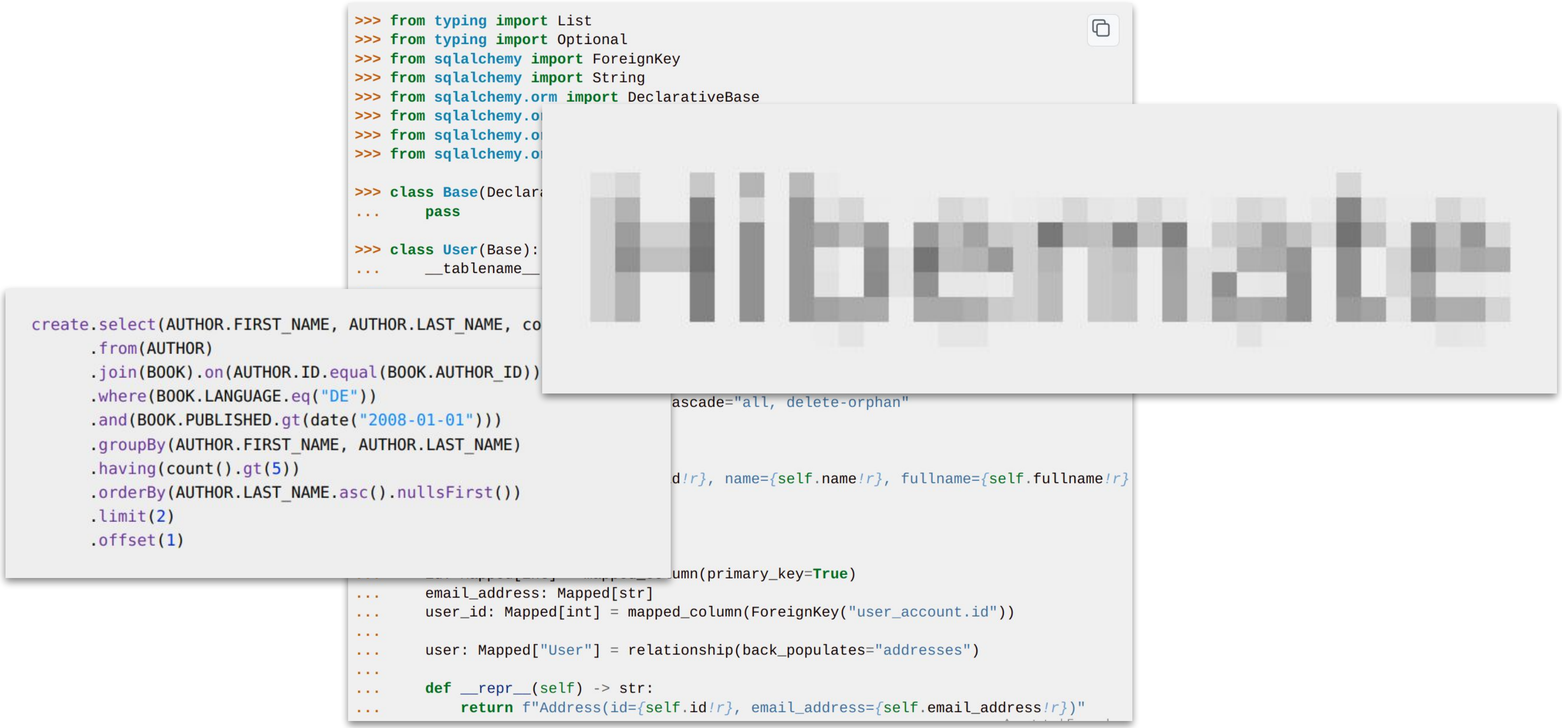
```
>>> from typing import List
>>> from typing import Optional
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy import String
>>> from sqlalchemy.orm import DeclarativeBase
>>> from sqlalchemy.orm import Mapped
>>> from sqlalchemy.orm import mapped_column
>>> from sqlalchemy.orm import relationship

>>> class Base(DeclarativeBase):
...     pass

>>> class User(Base):
...     __tablename__ = "user_account"
...
...     id: Mapped[int] = mapped_column(primary_key=True)
...     name: Mapped[str] = mapped_column(String(30))
...     fullname: Mapped[Optional[str]]
...
...     addresses: Mapped[List["Address"]] = relationship(
...         back_populates="user", cascade="all, delete-orphan"
...     )
...
...     def __repr__(self) -> str:
...         return f"User(id={self.id!r}, name={self.name!r}, fullname={self.fullname!r}

>>> class Address(Base):
...     __tablename__ = "address"
...
...     id: Mapped[int] = mapped_column(primary_key=True)
...     email_address: Mapped[str]
...     user_id: Mapped[int] = mapped_column(ForeignKey("user_account.id"))
...
...     user: Mapped["User"] = relationship(back_populates="addresses")
...
...     def __repr__(self) -> str:
...         return f"Address(id={self.id!r}, email_address={self.email_address!r})"
```



```
>>> from typing import List
>>> from typing import Optional
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy import String
>>> from sqlalchemy.orm import DeclarativeBase
>>> from sqlalchemy.orm import Mapped
>>> from sqlalchemy.orm import mapped_column
>>> from sqlalchemy.orm import relationship

>>> class Base(DeclarativeBase):
...     pass

>>> class User(Base):
...     __tablename__ = "user_account"

                                    umn(primary_key=True)
                                    olumn(String(30))
                                    r]]

                                    ess"]] = relationship(
                                    ascade="all, delete-orphan"



                                    d!r}, name={self.name!r}, fullname={self.fullname!r}




                                    umn(primary_key=True)
...     email_address: Mapped[str]
...     user_id: Mapped[int] = mapped_column(ForeignKey("user_account.id"))
...
...     user: Mapped["User"] = relationship(back_populates="addresses")
...
...     def __repr__(self) -> str:
...         return f"Address(id={self.id!r}, email_address={self.email_address!r})"
```



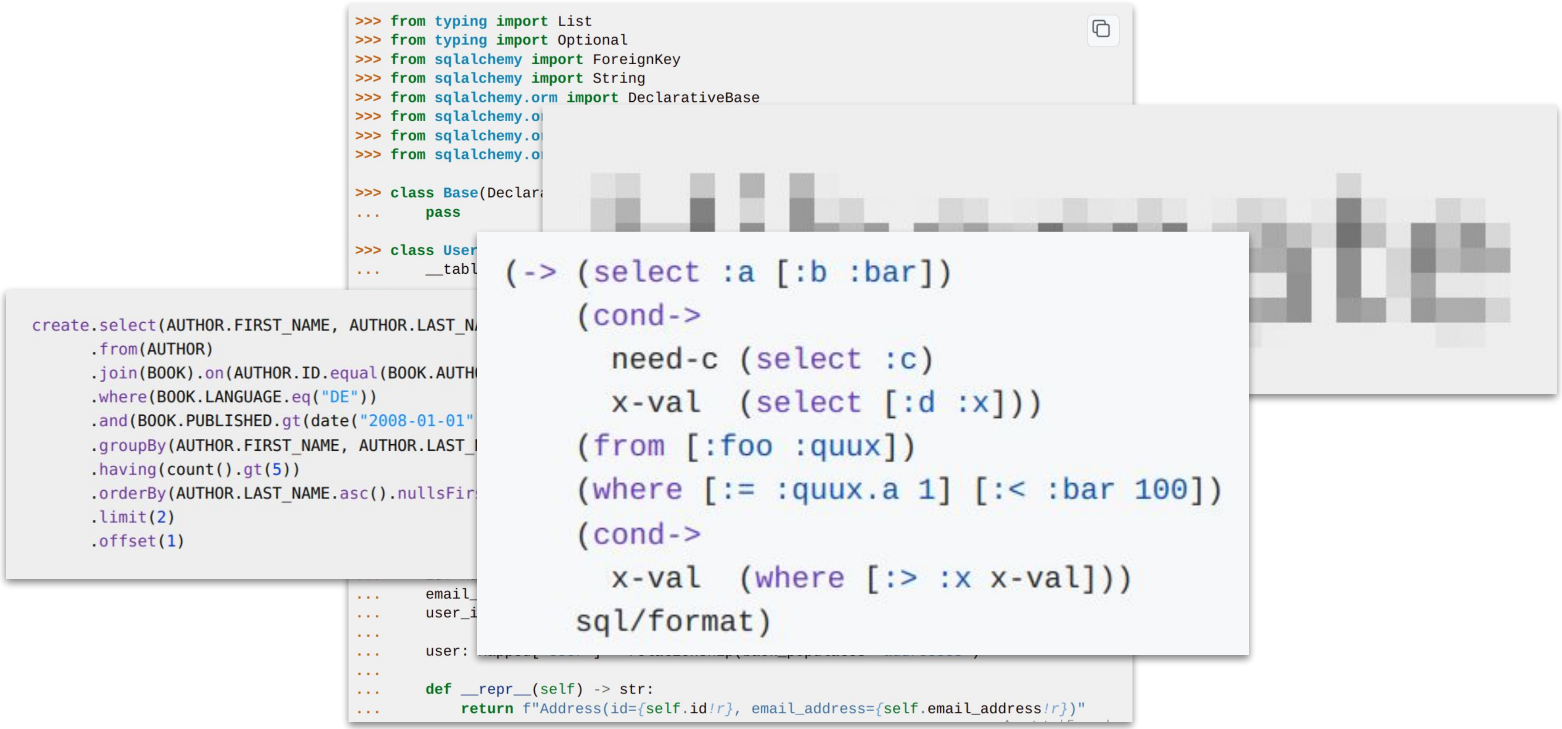
```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count())
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(2)
      .offset(1)
```



```
>>> from typing import List
>>> from typing import Optional
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy import String
>>> from sqlalchemy.orm import DeclarativeBase
>>> from sqlalchemy.o
>>> from sqlalchemy.o
>>> from sqlalchemy.o

>>> class Base(Declar
...     pass

>>> class User(Base):
...     __tablename__
```

```
ascade="all, delete-orphan"

d!r}, name={self.name!r}, fullname={self.fullname!r}

umn(primary_key=True)
...     email_address: Mapped[str]
...     user_id: Mapped[int] = mapped_column(ForeignKey("user_account.id"))
...
...     user: Mapped["User"] = relationship(back_populates="addresses")
...
...     def __repr__(self) -> str:
...         return f"Address(id={self.id!r}, email_address={self.email_address!r})"
```

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, co
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
      .limit(2)
      .offset(1)
```



```
>>> from typing import List
>>> from typing import Optional
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy import String
>>> from sqlalchemy.orm import DeclarativeBase
>>> from sqlalchemy.o
>>> from sqlalchemy.o
>>> from sqlalchemy.o

>>> class Base(Declar
...     pass

>>> class User
...     __tabl
...
...     email_
...     user_i
...
...     user:
...
...     def __repr__(self) -> str:
...         return f"Address(id={self.id!r}, email_address={self.email_address!r})"
```

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_N
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTH
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt(date("2008-01-01"
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFir
      .limit(2)
      .offset(1)
```

```
(-> (select :a [:b :bar])
    (cond->
      need-c (select :c)
      x-val  (select [:d :x]))
    (from [:foo :quux])
    (where [:= :quux.a 1] [:< :bar 100])
    (cond->
      x-val  (where [:> :x x-val]))
    sql/format)
```

☀️

```
mydb=> ▌
```

☀️

```
mydb=> ▌
```

```
~$ ▌
```

☀️

```
mydb=> ▌
```

```
~$ ▌
```

😊

🌤️

```
mydb=> ▌
```

```
~$ ▌
```

😊

```
~$ $EDITOR myscript.sh▌
```

⛅

```
mydb=> ▌
```

```
~$ ▌
```

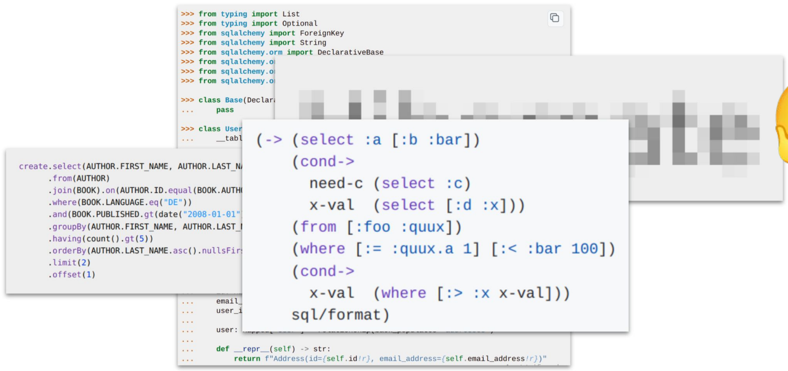
😊

```
~$ $EDITOR myscript.sh▌
```

🤔

```
mydb=> ▌
```

```
~$ ▌
```



```
>>> from typing import List
>>> from typing import Optional
>>> from sqlalchemy import ForeignKey
>>> from sqlalchemy import String
>>> from sqlalchemy.orm import DeclarativeBase
>>> from sqlalchemy.o
>>> from sqlalchemy.o
>>> from sqlalchemy.o

>>> class Base(Declar
...     pass

>>> class User
...     __tabl
...     email_
...     user_i
...
...     user:
...
...     def __repr__(self) -> str:
...         return f"Address(id={self.id!r}, email_address={self.email_address!r})"
```

```
create.select(AUTHOR.FIRST_NAME, AUTHOR.LAST_N
      .from(AUTHOR)
      .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTH
      .where(BOOK.LANGUAGE.eq("DE"))
      .and(BOOK.PUBLISHED.gt(date("2008-01-01"
      .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_
      .having(count().gt(5))
      .orderBy(AUTHOR.LAST_NAME.asc().nullsFir
      .limit(2)
      .offset(1)
```

```
(-> (select :a [:b :bar])
    (cond->
      need-c (select :c)
      x-val  (select [:d :x]))
    (from [:foo :quux])
    (where [:= :quux.a 1] [:< :bar 100])
    (cond->
      x-val  (where [:> :x x-val]))
    sql/format)
```

```
~$ $EDITOR myscript.sh▌
```

There is no such thing as “too many quotation marks”.
— Proverbial wisdom

```
def make_lookup_sql(columns, table, lookup_column, order_column, schema='public', confidential_columns=None):
    return sql.SQL(f"""
SELECT {",".join(['{}'] * len(columns))}
  FROM {{}}.{{}}
 WHERE {{}} = %s
 ORDER BY {{}} DESC
""").format(*[sql.Identifier(i)
              for i in (columns + [schema, table, lookup_column, order_column])])
```

```
def make_fetch_events_query(object_type):
    return sql.SQL("""
SELECT event_json
  FROM event_history.{}
 WHERE object_key = %(object_key)s
   AND  occurred_at > %(after)s::timestamp
   AND (occurred_at < %(until)s::timestamp OR (occurred_at = %(until)s::timestamp AND eid <= %(before_eid)s))
 ORDER BY occurred_at DESC, eid DESC
 LIMIT %(limit)s
    """).format(sql.Identifier(object_type))


def make_fetch_events_query_with_filter(object_type):
    return sql.SQL("""
SELECT event_json
  FROM event_history.{}
 WHERE object_key = %(object_key)s
   AND  occurred_at > %(after)s::timestamp
   AND (occurred_at < %(until)s::timestamp OR (occurred_at = %(until)s::timestamp AND eid <= %(before_eid)s))
   AND event_type = ANY(%(event_types)s)
 ORDER BY occurred_at DESC, eid DESC
 LIMIT %(limit)s
    """).format(sql.Identifier(object_type))
```

⛈️

```
def make_fetch_events_query(object_type):
    return sql.SQL("""
SELECT event_json
  FROM event_history.{}
 WHERE object_key = %(object_key)s
   AND  occurred_at > %(after)s::timestamp
   AND (occurred_at < %(until)s::timestamp OR (occurred_at = %(until)s::timestamp AND eid <= %(before_eid)s))
 ORDER BY occurred_at DESC, eid DESC
 LIMIT %(limit)s
    """).format(sql.Identifier(object_type))


def make_fetch_events_query_with_filter(object_type):
    return sql.SQL("""
SELECT event_json
  FROM event_history.{}
 WHERE object_key = %(object_key)s
   AND  occurred_at > %(after)s::timestamp
   AND (occurred_at < %(until)s::timestamp OR (occurred_at = %(until)s::timestamp AND eid <= %(before_eid)s))
   AND event_type = ANY(%(event_types)s)
 ORDER BY occurred_at DESC, eid DESC
 LIMIT %(limit)s
    """).format(sql.Identifier(object_type))
```

```
DEFAULT_AFTER_VALUE = '1990-01-01T00:00:00.000Z'
DEFAULT_UNTIL_VALUE = '2100-01-01T00:00:00.000Z'
```

```
def make_fetch_events_query(object_type):
    return sql.SQL("""
SELECT event_json
  FROM event_history.{}
 WHERE object_key = %(object_key)s
   AND  occurred_at > %(after)s::timestamp
   AND (occurred_at < %(until)s::timestamp OR (occurred_at = %(until)s::timestamp AND eid <= %(before_eid)s))
 ORDER BY occurred_at DESC, eid DESC
 LIMIT %(limit)s
    """).format(sql.Identifier(object_type))
```

```
def make_fetch_events_query_with_filter(object_type):
    return sql.SQL("""
SELECT event_json
  FROM event_history.{}
 WHERE object_key = %(object_key)s
   AND  occurred_at > %(after)s::timestamp
   AND (occurred_at < %(until)s::timestamp OR (occurred_at = %(until)s::timestamp AND eid <= %(before_eid)s))
   AND event_type = ANY(%(event_types)s)
 ORDER BY occurred_at DESC, eid DESC
 LIMIT %(limit)s
    """).format(sql.Identifier(object_type))
```

```
#define MACRO(args) \
    …
```

```
(defmacro macro [args]
    …
    )
```

🐲

```
#define MACRO(args) \
    ...
```

😣

```
(defmacro macro [args]
    ...
)
```

🐲

```
#define MACRO(args) \
    ...
```

😣

```
(defmacro macro [args]
    ...
)
```

😣

🤓

$$q' = f(q)$$

$$f\colon Q \rightarrow Q$$

🤓

$q' = f(q)$

$g\colon Q \to Q$

$f\colon Q \to Q$

$f \circ g\colon Q \to Q$

$$A, B \in R^{n \times n}$$

$$A \times B = C \in R^{n \times n}$$

The following is another striking example of the unobviousness of the scoping rules. Consider the following two queries:

```
SELECT SUM (QTY)                SELECT SUM (QTY)
FROM   SP                       FROM   SP
                                GROUP  BY P#
```

In the first case, the query returns a single value; the argument to the SUM invocation is the entire QTY column. In the second case, the query returns multiple values; the SUM function is invoked multiple times, once for each of the groups created by the GROUP BY clause. Notice how the meaning of the syntactic construct "SUM(QTY)" is dependent on context. In fact, SQL is moving out of the strict tabular framework of the relational model in this second example and introducing a new kind of data object, viz. a set of tables (which is of course not the same thing as a table at all). GROUP BY converts a table into a set of tables. In the example, SUM is then applied to (a column within) each member of that set. A more logical syntax might look something like the following:

```
APPLY ( SUM, SELECT QTY
             FROM ( GROUP SP BY P# ) )
```

Q7. Find the names of managers who manage more than ten employees.

$$x_{NAME} \in EMP : COUNT\ ({}_{NAME}EMP_{MGR}\ (x_{NAME}\ )) > 10$$

Note that in Q7 the free variable x is used to correlate a manager's name with a group of rows representing his employees, so that this group may be counted. Experience has shown that this sort of "grouping" occurs quite frequently in queries. Accordingly, a way is provided in SEQUEL to divide the rows of a table into groups according to the values of one or more columns, in a way analogous to the concept of a "glump" in Information Algebra (17). An optional GROUP BY clause may be attached to any FROM clause in SEQUEL, with the effect that the rows of the table are considered to be in groups of matching column-values. For example, if a query contains the clause FROM EMP GROUP BY MGR, the rows of the EMP table are formed into groups of matching MGR for the purpose of this query. Within the scope of such a clause, there are certain restrictions on the column-references which may be made. The grouping column or columns (MGR in the above example) may be referred to because it has only one value per group. Mathematical functions on column-values may be used, with the implied rule that they take a group of column-values as their argument and return a single value for the group. For example, within the scope of the clause FROM EMP GROUP BY MGR, the function AVG (SAL) would return, for each manager, the average salary of his employees. Other functions such as SUM, COUNT, and MAX operate in similar ways. A column-name which is not part of the grouping criterion may not be referred to except as an argument to a function which returns a single value per group. Example Q7 is now repeated in SEQUEL to illustrate the GROUP BY feature.

Q7. Find the names of managers who manage more than ten employees.

```
SELECT    MGR
FROM      EMP GROUP BY MGR
WHERE     COUNT (NAME ) > 10
```

🏁

1. 🤓
2. ...
3. 🏆

Frankly, there is so much confusion in this area that it is difficult to criticize it coherently.

… the argument is in fact specified in a most unorthodox manner,..

… functions are subject to a large number of peculiar and apparently arbitrary restrictions.

— A Critique of the SQL Database Language

📚

- Chamberlin, Donald D; Boyce, Raymond F (1974). "SEQUEL: A Structured English Query Language" (PDF). Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control.
- C. J. Date: A Critique of the SQL Database Language. SIGMOD Record 14(3): 8-54, 1984.
- C. J. Date, Hugh Darwen: Databases, Types and the Relational Model: The Third Manifesto. Addison-Wesley, 2007.