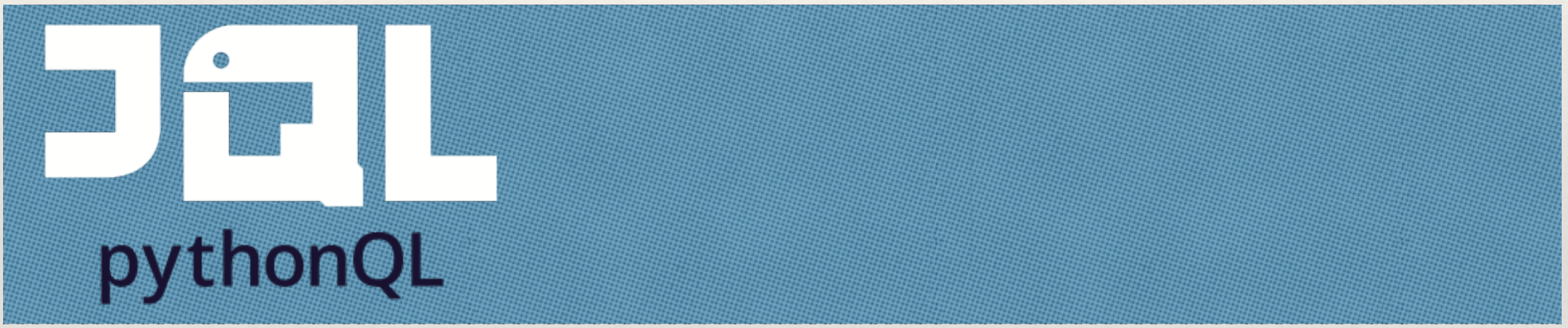


Встраивание языка запроса в Питон или
Почему  Data Scientists живут без SQL

# О чем будет доклад

- Наша мотивация
- Краткое введение в PythonQL
- Примеры сценариев, где особенно удобно пользоваться PythonQL
- Текущая реализация, планы на будущее

# Во-первых: команда

- Daniela Florescu
  - XQuery, JSONiq, Oracle, Zorba

- Pavel Velikhov
  - XQuery (Enosys,Sedna), SciDB, Data Science

- Sergey Vinogradov
  - Data Science

* все работы ведутся в свободное от основной работы время
** поэтому проект движется так неспешно

# Как это началось

- MongoDB, CouchDB :) Попытка как-то возродить JSONiq.
- Вывод: отдельный мощный язык для обработки данных делать неперспективно
- Посмотрели на Data Science:
  - Сообщество баз данных потеряло это аудиторию
  - Это по сути бывшие аналитики, которые постоянно работали с SQL
  - Теперь это люди, которые используют все что можно, но старательно обходят SQL. Почему?

# Data Science (I)

- Как обычно работает Data Scientist?
  - Куча разных источников данных, в разных форматах: CSV, JSON, текст (HDFS), разные базы данных, т.д.
  - Очень много задач класса: быстро оценить пользу от источника данных, быстро проверить гипотезу
  - Традиционная схема Data Integration (ETL) не работает
  - Вообще с базой данных не связываются, если:
    - Данные там уже не лежат
    - Объем не настолько огромной, что без СУБД не обойтись

# Data Science (II)

- Дополнительные аргументы против использования СУБД с мощным языком запросов:
  - Перед анализом данных, их надо чистить. Легче чистить данные на “родном” языке
  - Зачастую, результат анализа - качество построенной модели, а встроенный функционал по машинному обучению и статистке в СУБД сильно отстает. Плюс фактор “родного языка”

# PythonQL: Дизайн языка

- PythonQL состоит из нескольких компонент:
  - Основной язык: расширение comprehensions (list,set,map).
  - Путевые выражения (сверху JSON структур)
  - Try-except выражение
  - Конструктор кортежей

# PythonQL: Основной язык

- Похож и на SQL и на JSONiq, но является строгим супер-множеством Питона
- Минимальное расширение Питона, особенно это касается ключевых слов.
- По максимуму используются конструкции Питона, например нет своих кванторов существования и всеобщности, нет case statement, агрегатные и статистические функции из стандартных библиотек
- Самое большое расширение - это запросы с window. Функционал window взят из XQuery.

# PythonQL: Основной язык

query_expr := [ ‘select’ ] expr

( for | let | window)

( for | let | window | where | group by |

order by | count )*

Пример:

```
[ select (d.model, d.make, reviews)
  for d in dealer_db, p in product_db
  let p_make = try p.man[‘company’] except p.man
  where d.model = p.model and d.make = p.make
  let reviews = [ select r for r in p.reviews where r.stars == 5 ]
  where len(reviews) > 1 ]
```

# PythonQL: строгое супермножество comprehensions

- Comprehensions в Питоне:
  - [ (x,y)  **for** x **in** coll_1  **for** y **in** coll_2  **if** cond ]
  - { x  **for** x **in** collection  **if** cond }
  - { x.key : x.value  **for** x **in** collection  **if** cond }
  - ( x  **for** x **in** collection  **if** cond )

# PythonQL: Путевые выражения

- Путевые выражения очень хорошо себя показали в XQuery и JSONiq.
- Взяли пока минимальный набор из XQuery:
  - **for** x **in** object **./** ‘car’ **./** ‘make’
  - **for** x **in** object **../** _ **./** ‘make’
- Пока работают только сверху JSON (то есть list, set, dict объектов).
- Планируется поддержка XML

# PythonQL: маленькие приятности

- Try-except:
  - В Питоне механизм исключений реализован в statement, а не в expression
  - То есть внутри запроса нельзя пользоваться исключениями
  - добавили конструкцию: try expr except expr
  - Очень удобна при работе с слабо-структурированными и грязными данными, а также для быстрого анализа

# PythonQL: маленькие приятности

- Конструктор кортежей
- В Питоне есть свои кортежи, но они не именованные, доступ только по индексам

```
select (x.fname as fname, y.lname as lname)
for x in names
```

# Несколько сценариев использования (DataScience)

- Customer Journey: гетерогенные временные ряды событий о всех действиях клиента
- Data Cleaning & Integration: традиционный сценарий, но до сих пор тяжелый
- Model Evaluation: интересный сценарий, метод, который предлагаем мы сейчас почти не используется

# Customer Journey

- Путешесвие клиента через все стадии услуги
- Разнородные события:
  - открыл счет, перевод денег, заявка на кредит, выдача кредита, выплата кредита, звонок из кол-центра, закрытие счета
- Нужна разнородная ad-hoc аналитика, быстрая проверка гипотез

# Customer Journey

Сколько клиентов с балансом > 300 долларов в разных штатах?

```
res = [
    # Пробежимся во всем клиентам (клиент - это список событий)
    select (state, len(balance) as n_customers)
    for cj in cust_journeys

 # Выделим данные о клиенте из события об открытие счета
let c_data = [select e for e in cj where e.event_name==‘open’][0].client_data,

    # Достанем суммы транзакцию по внесению и снятию денег со счета
    withdrawals = [select e.amount for e in cj where e.event_name=='withdraw'],
    deposits = [select e.amount for e in cj where e.event_name=='deposit']

# Посчитаем баланс и отфильтруем клиентов
let balance = sum(deposits) - sum(withdrawals)
where balance > 300

# Сгруппируем по штату
group by c_data.address.state as state ]
```

# Customer Journey

Клиенты, которым было отказано в кредите, и которые закрыли счета в течение месяца после этого события.

```
n_closed_and_refused = [
    # Пробежимся по всем клиентам (клиент = список событий)
    select cj for cj in cust_journeys
    # Найдем событие закрытия счета, если такого нет, пропустим этого клиента
    let close = next((select e for e in cj where e.event_name=='close'),None)
    where close

    # Узнаем дату последней заявки на кредит, если заявки не было, пропустим клиента
    let req = try [select e for e in cj where e.event_name==‘loan_req’][-1] except None
    where req
    let last_request_date = parse(req.date),
        close_date = parse(close.date)

    # Если кредит не выдали и заявка была за месяц или раньше - это наш клиент
    where (close_date - last_request_date).days < 30
      and not [select e for e in cj
               where e.event_name=='loan_issued' and
               (parse(e.date) - last_request_date) > 0 ]  ]
```

# Data Integration & Cleaning

- Здесь целый спектр проблем, но мы фокусируемся на быстрой проверке гипотез.
- Задача - оценить источник данных максимально быстро
- Например у нас есть база клиентов, и мы нашли базу ритейлеров, которая способна обогатить нашу основную базу
- Мы проверяем, насколько она будет полезна - то есть насколько хорошо состыкуются эти базы

# Data Integration & Cleaning

```
joined_local_orders = ([
  # Попробуем сделать слияние двух баз
  select amount   for m in master_db, o in order_db

  # Применим soundex для игнорирования мелких ошибок написания
  where soundex(m.first_name) == soundex(o.first_name) and
        soundex(m.last_name) == soundex(o.last_name) and

        # Но нас интересуют покупки по месту жительства
        any([ select c_addr.city == o.store.address.city
              for c_addr in m.addresses ])

  # Попробуем еще пропарсить сумму покупки, если не получается
  # то выкидываем эти данные
  let amount = try Decimal(o.amount) except None
  where amount ])
```

# Model Evaluation

- Выбор и оценка модели - очень типичная задача для машинного обучения
- Мы рассмотрим сценарий, где сравниваются все модели и надо понять в каких случаях и насколько сильно различаются их результаты
- Возьмем такой пример: у нас есть данные о домах и две модели, которые предсказывают цену дома

# Model Evaluation

```
res = [
  # В результате мы хотим получить разбивку по городам, где видна корреляция моделей,
  # а также средние квадратичные ошибки моделей
  select (city, corr, mrsq_1, mrsq_2)

  # Соберем воедино данные
  for d in dataset, m_1 in model1, m_2 in model2
  where d.record_id == m_1.record_id and d.record_id == m_2.record_id

# Отдельно выделим атрибуты, которые превратятся в списки после группировки
let price = d.price, m1_pred = m_1.pred, m2_pred = m_2.pred

# Сгруппируем по городу
group by d.city as city

# Вычислим ошибки и корреляция методами Питона (sklean, scipy)
let mrsq_1 = mean_squared_error(price, m1_pred),
    mrsq_2 = mean_squared_error(price, m2_pred),
    corr = pearsonr(m1_pred, m2_pred)[0]

# Отсортируем по корреляции по убыванию
order by corr desc ]
```

# Текущая реализация

- Реализация похожа на препроцессор, PythonQL транслируется в Python и исполняется обычным Питоном
- Грамматический разбор реализован на ANTLR4:
  - Разбор получился очень медленным
  - Сообщения об ошибках низкого качества
- Примитивный процессор запросов, написанный на Питоне. Все внутренние выражения компилируются и исполняются через eval.
- Нет оптимизатора и планировщика запросов

# Дальнейший план:

- Установка пакета через стандартный менеджер пакетов
- Использование pythonql через механизм кодировки кода в Питоне
  - #coding: pythonql
- Рудиментарный wrapper для SQL баз, Apache Spark

# Долгосрочные вызовы

- Оптимизатор и планировщик
  - Все обычные проблемы разработки медиатора
  - Дополнительные неприятности из-за динамичности языка
    - Например: в Питоне без проблем внутри любой функции можно переопределить функции любых библиотек
    - Как бороться:
      - Создать аннотации, гарантирующие что функция хорошая
      - Перед запуском запроса проверить, что функции не переопределены (через интроспекцию)

# Долгосрочные вызовы

- Процессор запросов:
  - Нужна эффективная реализация на C++ для много-ядерного исполнения
  - Переписывание и оптимизация Apache Spark планов для параллельного исполнения
- Поддержка разных форматов результатов запросов (а значит и внутренние представления):
  - Просто список кортежей в Питоне: очень некомпактное представление
  - Массивы numpy (C++ массивы): компактное представление + векторные операции. Но не гетерогенные
  - pandas Dataframe - реализованы сверху numpy

# Вопросы и ответы

- Спасибо за внимание
- Давайте обсудим!