ChatGPT

SIMD и SISD: теория и практика

Архитектуры CPU по Флинну: SISD vs SIMD

SISD (Single Instruction, Single Data) – архитектура, при которой процессор за такт выполняет **одну команду над одним элементом данных**. Классический однопоточный CPU соответствует типу SISD. Пример: обычный цикл `for` на Python – на каждой итерации выполняется одна операция над одним значением и результат получается шаг за шагом [1]. Такой подход очень универсален (позволяет выполнять произвольные операции), но неэффективен для больших объемов данных, так как каждый элемент обрабатывается последовательно.

SIMD (Single Instruction, Multiple Data) – архитектура, при которой **одна команда выполняется сразу над несколькими данными**. Современные процессоры содержат специальные векторные регистры и блоки SIMD, позволяющие за один такт применять операцию сразу к набору из нескольких значений [2]. Ширина таких регистров обычно 128, 256 или 512 бит, что даёт ускорение, зависящее от типа данных (сколько чисел помещается в вектор). Например, если регистр 256-битный и хранит по 32 бита на число, то одна команда может обработать 8 чисел одновременно. **SIMD обеспечивает параллельную обработку на уровне данных** и существенно ускоряет простые однотипные операции на больших массивах. В контексте Python SIMD задействуется не в самом интерпретаторе, а в низкоуровневых библиотеках: так, векторизированные операции в NumPy и Polars используют SIMD внутри нативного кода [1] [3].

Пример: последовательная vs векторизованная обработка в Python

Рассмотрим простой пример сложения элементов двух массивов, реализованный двумя способами – **последовательно (SISD)** с помощью чистого Python и **векторизованно (SIMD)** с помощью NumPy:

```
import numpy as np

# Данные: два списка из 10 миллионов чисел
N = 10_000_000
list_a = list(range(N))
list_b = list(range(N))

# 1. Складываем списки поэлементно в чистом Python (SISD)
list_c = []
for a, b in zip(list_a, list_b):
    list_c.append(a + b)

# 2. Складываем массивы с помощью NumPy (SIMD под капотом)
arr_a = np.array(list_a, dtype=np.int64)
```

```
arr_b = np.array(list_b, dtype=np.int64)
arr_c = arr_a + arr_b
```

В первом случае Python выполняет миллион итераций цикла `for`, на каждой выполняя одну инструкцию сложения над одной парой чисел – типичный SISD-подход. Во втором случае выражение `arr_a + arr_b` делегирует работу высокопроизводительному C-коду NumPy, который внутри одним вызовом обрабатывает сразу большой блок данных. Например, NumPy может воспользоваться SIMD-инструкциями процессора, выполняя **одну машинную инструкцию над несколькими элементами массива одновременно**. В результате второй вариант работает во много раз быстрее первого, особенно на больших массивах (ускорение в десятки и сотни раз не редкость).

Почему такой выигрыш в скорости? При чисто Python-цикле накладные расходы интерпретатора на каждую итерацию очень высоки – требуется обработать байткод, выполнить операции над Python-объектами. К тому же, без SIMD процессор на каждой операции задействует только часть своих ресурсов. В случае NumPy вся основная работа выполнена на уровне Си: цикл сложения реализован внутри низкоуровневой функции и оптимизирован компилятором. Компилятор может *автовекторизовать* такой цикл, задействуя SIMD-регистр и складывая, например, по 4 или 8 чисел за итерацию. Таким образом, **векторизация (SIMD) позволяет уменьшить число инструкций CPU, обрабатывая несколько данных за команду**, и снижает интерпретаторские накладные расходы [1]. Это демонстрирует принцип: для вычислительно интенсивных задач предпочтительно использовать векторизированные библиотеки (NumPy, Pandas, Polars), а не чистые Python-циклы.

SIMD в реализации NumPy и Polars

Библиотека **NumPy** изначально написана на C и способна эффективно использовать возможности аппаратуры. Многие операции над `ndarray` в NumPy выполняются внутри высокооптимизированных функций (напрямую в C или с использованием библиотек BLAS). Компиляторы (gcc, clang) обычно автоматически векторизуют простые циклы. Например, операции над массивами (`+`, `*`, вычисление экспоненты и др.) компилируются в машинный код, который задействует SIMD-регистры, если это возможно. Кроме того, для линейной алгебры NumPy может использовать оптимизированные реализации (OpenBLAS, MKL), где SIMD-инструкции (SSE, AVX и т.д.) используются вручную. В результате такие функции как умножение матриц, сортировка, агрегаты и др. работают на скоростях, близких к языкам C/Fortran, существенно обгоняя чистый Python. Например, известно, что даже операция суммирования в NumPy для больших массивов реализована не простым Python-циклом, а специальным методом (pairwise summation) и с недавнего времени поддерживает SIMD для целочисленных массивов (AVX2 256-бит) [4]. Таким образом, **NumPy активно использует SIMD где возможно**, хотя детали зависят от конкретной версии и настроек компиляции.

Polars – современная колоночная библиотека на Rust – также широко применяет SIMD. Polars основан на формате памяти Apache Arrow и своих алгоритмах на Rust. Благодаря *колоночному* представлению данных (каждый столбец хранится в непрерывном массиве в памяти) легко применять векторные инструкции для обработки целых колонок. Документация отмечает, что Polars и его бекенд Arrow используют SIMD для оптимизации внутренних вычислительных ядер [3]. В частности, в Polars есть опциональная поддержка пакета `packed_simd` (включается при компиляции с nightly-компилятором Rust), ускоряющая многие операции над столбцами [5]. Даже без специальных флагов, Arrow и Polars стараются выровнять данные по границам и обрабатывать их блочно, чтобы **максимально задействовать SIMD-ускорение процессора**.

Например, операции фильтрации, агрегирования по столбцу, вычисления статистик реализованы с учетом возможностей векторизации.

Важно понимать, что высокоуровневый Python-код (например, методы Pandas/Polars) не требует от пользователя ручного использования SIMD – это происходит автоматически на уровне реализации библиотеки. То есть, когда вы пишете `df["col"] + 1` в Pandas или Polars, внутри для целочисленного столбца выполнится цикл на C/Rust, который, скорее всего, пройдет оптимизацию и будет работать по несколько элементов за раз.

Анализ SIMD: профилировка и мониторинг

Как убедиться, что под капотом действительно работают SIMD-инструкции? Этот процесс невидим на уровне Python-кода, но существуют инструменты для низкоуровневого анализа производительности:

- **Профилирование производительности CPU (Linux `perf`, Intel VTune):** С помощью профилировщиков, которые умеют собирать аппаратные счётчики, можно выяснить, какие инструкции выполняются. Например, `perf` способен подсчитать количество выполненных SIMD-инструкций (таких как инструкции SSE/AVX) во время выполнения вашего кода [6]. Аналогично, Intel VTune умеет показывать долю векторизированных операций (Packed vs Scalar) и выявлять, где код не использует SIMD. Эти инструменты требуют знаний низкого уровня и умения интерпретировать assembler и hardware counters, но дают наиболее точную информацию.

- **Дизассемблирование кода:** Можно посмотреть машинный код функций, реализующих ту или иную операцию. Для Python-функций, написанных на C (например, ufunc в NumPy), можно с помощью утилиты `objdump` или аналогов увидеть, присутствуют ли там инструкции SSE/AVX [6]. Например, определить, вызывает ли `np.add` SIMD-инструкции, можно найдя в ассемблерном листинге команды типа `addps` (сложение пакетов по SIMD) и т.п. Однако это трудоемко: нужно знать точное имя C-функции и понимать ассемблер.

- **Модули Python для отладки:** Сам модуль `dis` (дизассемблер байткода Python) показывает лишь байткод интерпретатора (например, цикл for покажет набор операций LOAD_FAST, BINARY_ADD и т.д.), но не раскрывает внутренних оптимизаций NumPy. Вместо него можно использовать профилировщики уровня Python (например, `py-spy`, `cProfile`) чтобы убедиться, что основное время выполняется внутри низкоуровневых функций (что косвенно указывает на их эффективность). **py-spy** полезен для определения "горячих точек" – например, он покажет, что при векторизированном коде почти все время уходит в вызов функции NumPy (написанной на C), а не растрачивается на интерпретатор Python.

- **Специальные бенчмарки:** В качестве практического подхода можно сравнить время работы вычислений с разным размером данных. Если время растет линейно и с небольшой константой – это признак оптимизированного (возможно SIMD) кода. Для ещё более явного эффекта можно попробовать отключить определенные расширения CPU (например, запустить программу с переменной окружения, запрещающей AVX), и увидеть разницу во времени.

Подводя итог: обнаружить использование SIMD напрямую не всегда просто, но косвенные признаки – высокая производительность при больших объемах данных, низкая загрузка одного

ядра при расчётах (т.к. за такт обрабатывается много данных), а также выводы инструментов профилирования – подтверждают, что **библиотеки вроде NumPy/Polars действительно задействуют возможности современных CPU**.

Обзор библиотек NumPy, Pandas и Polars

NumPy: архитектура и основные возможности

NumPy (Numerical Python) – фундаментальная библиотека Python для научных вычислений, предоставляющая **N-мерный массив** (`numpy.ndarray`) и богатый набор функций для работы с такими массивами. Ключевая особенность NumPy – **компактное хранение и обработка однотипных данных**. В массиве все элементы одного типа (int32, float64 и т.д.) хранятся подряд в памяти, без накладных объектов Python, что обеспечивает высокую эффективность. Архитектура NumPy ориентирована на делегирование вычислений оптимизированному коду на C/Fortran. Основные компоненты и принципы работы NumPy:

- **Базовый объект ndarray:** это обертка вокруг непрерывного блока памяти, содержащего элементы одинакового типа. ndarray хранит метаданные (размеры, тип, шаги) и указатель на данные. Операции над массивами часто реализованы как C-функции, которые получают указатель на данные и выполняют цикл по элементам.

- **Универсальные функции (ufunc):** Это векторизованные функции, такие как `np.add`, `np.sin`, `np.maximum` и т.д., которые применяются поэлементно к массивам. Ufunc написаны на C и автоматически обрабатывают массивы любых совместимых размеров (с учетом **broadcasting** – механизма расширения размеров). При вызове, например, `np.add(arr, 5)` внутренняя функция обойдет массив `arr` и прибавит 5 ко всем элементам, используя оптимальные инструкции. Ufunc могут быть бинарными (принимают два массива) или унарными (один массив). Они часто распараллеливаются на уровне SIMD, как обсуждалось выше.

- **Линейная алгебра и FFT:** NumPy включает модуль `numpy.linalg` и прочие, но обычно полагается на высокопроизводительные библиотеки (BLAS/LAPACK) под капотом. Например, умножение матриц (`np.dot` или `@`) вызывает BLAS-реализацию (ATLAS, OpenBLAS, Intel MKL и т.д.), которая использует многопоточность и векторизацию.

- **Интеграция с C/Fortran:** NumPy предоставляет API для написания собственных модулей на C, работающих с массивами напрямую. Это позволяет расширять функциональность без потери производительности.

Основные возможности NumPy включают быстрое чтение/запись массивов, индексацию и **срезы (slicing)** без копирования, **булеву фильтрацию** (`arr[arr > 0]`), агрегаты (`arr.sum()`, `arr.mean()`), линейную алгебру (`np.dot`, `np.linalg.svd`), генерацию случайных чисел и многое другое. Все эти операции реализованы с упором на эффективность.

Пример использования NumPy: вычислим среднеквадратичное отклонение массива – сначала “вручную” циклом, затем с помощью NumPy:

```python
import math, time
import numpy as np

data = list(range(1000000))          # Python-список
# Вычисление std вручную (SISD)
start = time.time()
mean = sum(data) / len(data)
variance = sum((x - mean)**2 for x in data) / len(data)
std_manual = math.sqrt(variance)
print("Std (manual):", std_manual, "Time:", time.time()-start)

# Вычисление std с NumPy (SIMD внутри)
arr = np.array(data, dtype=np.float64)
start = time.time()
std_numpy = arr.std()                # встроенная функция NumPy
print("Std (NumPy):", std_numpy, "Time:", time.time()-start)
```

На выходе (в зависимости от мощности CPU) второй вариант будет в десятки раз быстрее. Например, NumPy может занять ~0.01 с против ~0.2-0.3 с у чистого Python. Эта разница демонстрирует силу векторизации. **Вывод:** NumPy – это *базовый инструмент для численных данных* в Python, предоставляющий эффективность C при удобстве Python, за счет продуманной архитектуры массивов и использования SIMD.

Pandas: архитектура и основные возможности

Pandas – популярная библиотека для анализа данных, построенная поверх NumPy. Она вводит высокоуровневые структуры: **Series** (одно измерение, аналог столбца или массива с индексом) и **DataFrame** (таблица, представляющая собой коллекцию Series). Архитектура Pandas ориентирована на удобство работы с табличными данными (как в статистических пакетах или R data.frame), предлагая богатые средства группировки, соединения, времени и др. Ключевые особенности архитектуры Pandas:

- **DataFrame = набор колонок Series:** Внутренне DataFrame хранит каждую колонку как `pd.Series`. А каждая Series, в свою очередь, обычно хранит данные в `numpy.ndarray`. То есть Pandas по большей части использует **массивы NumPy для хранения числовых и некоторых других типов**. Исключение – колонки типа `object` (например, строки), которые хранятся как массивы указателей на Python-объекты (что неэффективно по памяти, но универсально). С выходом Pandas 2.0 появилась возможность хранить данные в колонках как типы Arrow (через бекенд PyArrow), но по умолчанию Pandas <2.x = NumPy-массивы [7].

- **Индекс и выравнивание по меткам:** В отличие от Polars, Pandas использует **индекс** – специальный объект для меток строк. Индекс может быть простым range (0,1,2,...) или сложным (даты, многокомпонентный). Индекс влияет на операции: например, при арифметике или объединении DataFrame происходит выравнивание по индексам. Это удобная функциональность, но добавляет накладные расходы. Polars, напротив, не имеет индексов – строки адресуются по позиции [8].

- **Монолитная однопоточная реализация:** Основные операции Pandas (например, `df.groupby().agg()`, `df.merge()`) реализованы на Python/Cython или используют NumPy. Pandas не прозрачно многопоточна – внутри одного вызова обычно задействуется только одно ядро CPU. Есть некоторые исключения: например, чтение CSV может использовать несколько потоков, некоторые низкоуровневые части (сортировка, perhaps) могут, но в целом **Pandas однопоточна**, и для параллелизма требуется внешняя помощь (например, Dask для распределения по ядрам) [9]. В Pandas 2.0 при использовании PyArrow-бекенда часть операций (например, арифметика над колонками Arrow) может делегироваться многопоточным алгоритмам Arrow, но это пока частные случаи.

- **Богатый API для табличных данных:** Pandas славится многочисленными функциями: удобное **чтение/запись** (CSV, Excel, SQL, JSON и др.), **индексирование и выборка** (`loc`, `iloc`, маски, сложные индексы), **группировка и агрегирование** (`groupby`), **слияния и объединения** (`merge`, `join`), **работа с пропущенными значениями**, **временные ряды** (частотные преобразования, скользящие окна), **категориальные данные**, **сводные таблицы**, и многое другое. Многие функции Pandas реализованы на чистом Python/Pandas (с Cython) и не все оптимизированы так же, как NumPy по скорости. Однако, за счёт удобства и гибкости Pandas стала де-факто стандартом для аналитиков.

- **Модель копирования/представления:** Pandas старается следовать семантике копирования при большинстве операций (чтобы изменения не влияли на оригинальные данные, если не указано иначе). Это упрощает понимание, но иногда приводит к избыточному копированию и расходу памяти. Существует механизм `view` (представление на тех же данных) для некоторых случаев (срезы, NumPy-колонки), но он ограничен.

Пример использования Pandas: посчитаем средний объем продаж по категориям товаров:

```
import pandas as pd

# Создаем DataFrame
data = {
    "Категория": ["A", "B", "A", "C", "B", "A"],
    "Продажи": [100, 200, 150, 300, 120, 180]
}
df = pd.DataFrame(data)

# Группировка по категории и расчёт средней продажи
mean_sales = df.groupby("Категория")["Продажи"].mean()
print(mean_sales)
```

Результат будет Pandas Series со средними значениями для A, B, C. В данном примере Pandas достаточно быстр, так как объем данных мал. **Однако на миллионах строк группировка Pandas может работать медленно из-за GIL и ограничений однопоточности.** В таких случаях и появляются преимущества альтернатив (как Polars).

Polars: архитектура и основные возможности

Polars – новая высокопроизводительная библиотека для работы с данными (DataFrame-структура), написанная на Rust. Polars можно рассматривать как современную альтернативу Pandas, учитывающую уроки архитектуры последних лет (колоночное хранение, многопоточность, ленивые вычисления). Ключевые черты архитектуры Polars:

- **Apache Arrow в памяти:** Polars хранит данные в формате, совместимом со спецификацией Apache Arrow [7]. Arrow – колончно-ориентированный формат, оптимизированный для аналитических задач. Это означает, что каждый столбец (`pl.Series` в Polars) хранится как массив значений одного типа + вспомогательные буферы (например, битовая маска валидности для обозначения null). Такое хранение повышает **локальность данных в кэше** и снижает накладные расходы на пропуски (отдельный бит на null). В Arrow/Polars чётко различаются **отсутствующее значение** и просто специальное значение (в Pandas часто `NaN` используется и для float, и для обозначения отсутствия). Колоночный формат идеально подходит для SIMD-обработки и сжатия данных [3].

- **Отсутствие индекса:** В Polars **нет скрытого индекса у строк** – каждая строка просто имеет позиционный индекс 0..N-1. Это упрощает модель данных: операций выравнивания по индексу нет, все объединения явные по ключам. Такой подход уменьшает вероятность ошибок (нет ситуации рассогласования индексов) и упрощает внутренние оптимизации. Если нужен классический индекс для ускорения поисков, Polars может использовать индексы как вспомогательную структуру (аналогично реляционным БД) [10], но не меняет семантику запросов.

- **Многопоточность и параллелизм:** Благодаря тому, что Polars написан на Rust и использует внутренне безопасный параллелизм, многие операции выполняются с задействованием всех ядер процессора. **Polars по умолчанию многопоточен**, в отличие от Pandas [9]. Например, группировка, сортировка, агрегации в Polars будут автоматически распараллелены: столбцы или части данных распределяются по потокам. Это дает почти линейный выигрыш на многоядерных машинах. Также Polars эффективнее использует SIMD (что мы обсуждали) и избегает GIL, так как основная работа – вне Python.

- **Ленивое исполнение (Lazy API):** Важнейшая особенность Polars – поддержка **ленивых вычислений**. По умолчанию Polars предоставляет *жадный API* (eager), где операции выполняются сразу (как в Pandas). Но пользователь может переключиться на *ленивый режим*, построив цепочку операций без их немедленного выполнения, а затем выполнить *весь запрос целиком*. Ленивое исполнение позволяет движку Polars оптимизировать запрос перед запуском (подробнее в следующем разделе). Многие высокоуровневые методы Polars имеют реализацию через ленивый движок. Например, если вызвать `df.join(other_df)` на больших таблицах, Polars может под капотом проанализировать запрос и оптимизировать его выполнение.

- **Близость синтаксиса к Pandas, но с расширениями:** API Polars во многом напоминает Pandas, что облегчает переход. Операции чтения данных (`pl.read_csv`, `pl.read_parquet`), фильтрация (`df.filter(pl.col("x") > 0)`), группировка (`df.groupby("key").agg([...])`), объединения (`df.join(other)`) – все это есть. Однако есть отличия: нет индекса, некоторые методы иначе называются (например, вместо `df.apply()` используются выражения), отсутствуют некоторые удобства Pandas (время покажет, будут ли добавлены). Polars предлагает **экспрессии (expressions)** – это

функциональный стиль обработки колонок, позволяющий лаконично описывать трансформации. Пример: `df.select(pl.col("a") * 2)` умножит столбец `a` на 2. Выражения особенно мощны в ленивом режиме.

Пример использования Polars (eager): решим ту же задачу, что и в Pandas, – средние продажи по категориям:

```
import polars as pl

data = {
    "Категория": ["A", "B", "A", "C", "B", "A"],
    "Продажи": [100, 200, 150, 300, 120, 180]
}
df_pl = pl.DataFrame(data)

mean_sales = df_pl.groupby("Категория").agg(pl.col("Продажи").mean())
print(mean_sales)
```

Вывод Polars будет DataFrame примерно такого вида:

```
shape: (3, 2)
┌────────────┬──────────┐
│ Категория  ┆ Продажи  │
│ ---        ┆ ---      │
│ str        ┆ f64      │
╞════════════╪══════════╡
│ A          ┆ 143.3333 │
│ B          ┆ 160.0    │
│ C          ┆ 300.0    │
└────────────┴──────────┘
```

Как видим, синтаксис очень похож на Pandas, разве что агрегирование оформлено через выражение `pl.col(...).mean()`. Этот запрос выполняется сразу (eager) и результат доступен в `mean_sales`. На небольших данных разница с Pandas незаметна, но на больших Polars будет работать быстрее благодаря архитектуре. В следующем разделе мы подробно сравним производительность Pandas и Polars.

Сравнение производительности Pandas и Polars

Одно дело – теоретические преимущества Polars, другое – реальные измерения. Рассмотрим типовые операции (фильтрация, агрегирование, группировка) и сравним скорость Pandas и Polars на одинаковых данных.

Примечание: Pandas 2.0 представил экспериментальную поддержку Arrow (через `dtype="arrow"` для колонок), но в большинстве сценариев Pandas по-прежнему уступает Polars.

Даже с Arrow-бекендом Pandas 2.0 часто медленнее, чем классический Pandas на NumPy, и тем более Polars [11]. Поэтому будем считать Pandas = классический режим (NumPy).

Фильтрация (where) на большом наборе данных

Предположим, у нас таблица с миллионом записей, и мы хотим отобрать строки по условию (например, `df["A"] > 500`). В Pandas это делается как `df[df["A"] > 500]`, в Polars – `df.filter(pl.col("A") > 500)`. В обоих случаях результат – отфильтрованный DataFrame. Однако время выполнения будет разным. Тесты показывают, что **Polars выполняет числовую фильтрацию в несколько раз быстрее**. В одном из сравнений Polars опережал Pandas примерно в 2–5 раз на фильтрации по числовому столбцу [12]. Наш эксперимент с случайными данными (1 млн строк) дал аналогичный результат: фильтрация Pandas ~0.06 с, Polars ~0.009 с (почти **6-кратное** превосходство Polars).

Почему такая разница? Pandas при фильтрации создает булев массив (`Series` типа bool) и затем применяет его к DataFrame, копируя соответствующие данные. Этот процесс однопоточный и использует NumPy, но ограничен скоростью памяти и работы одного ядра. Polars же выполняет фильтрацию **параллельно** на нескольких ядрах и сразу формирует новый DataFrame. Благодаря колоночному формату, условие проверяется очень эффективно (возможно, SIMD-поблочно), и нужные строки извлекаются. Также Polars оптимизирует работу с памятью – вместо создания копий колонок может переиспользовать буферы или отложить материализацию до необходимости.

Отметим, что на **фильтрации по строковым данным** (например, `df[df["Category"] == "A"]`) Pandas особенно медленен, если строки хранятся как `object` (каждая строка – Python-объект). Polars хранит строки как UTF8 в Arrow формате, что гораздо эффективнее. Поэтому при фильтрах по категориям/строкам Polars выигрывает ещё больше. В источниках отмечено, что **Pandas значительно уступает Polars на строковых фильтрах**, особенно на больших объемах данных [12].

Группировка и агрегирование

Группировка (`groupby`) – операция, где Pandas традиционно испытывает трудности на больших данных. Проверим задачу: для набора из N миллионов строк сгруппировать по одной или нескольким колонкам и посчитать, скажем, среднее по другой колонке. Pandas выполнит это с использованием одного ядра (возможно, с некоторой оптимизацией на Cython), а Polars разделит данные между потоками и использует все ядра.

Практические бенчмарки подтверждают, что **Polars существенно быстрее Pandas при группировках**, особенно если число групп велико или группировка многоключевая. В одном тесте на ~5 миллионов строк с группировкой по одной колонке Polars работал примерно в 3-4 раза быстрее. С увеличением числа столбцов для группировки производительность Pandas резко падает: при группировке по 5 колонкам Pandas 2.0 (Arrow) вообще не справился за разумное время (>1000 с), тогда как Polars выполнил за секунды [13] [14]. Даже Pandas 2.0 с NumPy-бекендом на сложных группировках значительно медленнее Polars.

Еще пример: расчет агрегатов (min, max, mean) для нескольких столбцов. Pandas нужно явно перечислить агрегаты или использовать `.agg()` с списком, Polars позволяет одним выражением применить несколько функций сразу. В эксперименте Polars и Pandas 2.0 показали близкую скорость на тривиальных агрегациях [15], но по мере усложнения (несколько функций,

несколько столбцов) Polars берёт верх. Причина – Polars выполняет все агрегаты в одном проходе (благодаря локальному оптимизатору), а Pandas может выполнять их по отдельности или с большим количеством Python-логики.

Вывод: для операций группировки/агрегирования **Polars почти всегда быстрее**. Он эффективнее масштабируется на многоядерных системах и написан на языке системного уровня (Rust) без GIL. Pandas же вынужден либо работать последовательно, либо привлекать сторонние решения для параллелизма. Источники едины во мнении: *"Polars – лучший выбор для группировок и агрегаций"* [11], опережая Pandas на порядок в экстремальных случаях. В то же время, на очень маленьких наборах (тысячи строк) разница может быть несущественной – иногда Pandas даже быстрее из-за меньших накладных расходов. Но при росте данных Polars выигрывает.

Причины различий в производительности

Суммируем технические причины, почему **Polars опережает Pandas** на многих задачах:

- **Многопоточность:** Polars изначально спроектирован для параллельной работы и использует все доступные ядра. Pandas преимущественно однопоточный (за редкими исключениями), поэтому Polars получает ~N-кратный выигрыш на N-ядрах для CPU-ограниченных задач [9].

- **Отсутствие GIL:** В Polars тяжелые вычисления происходят вне Python (Rust), поэтому глобальная блокировка интерпретатора (GIL) не мешает параллельным потокам. Pandas же даже в Cython-реализациях часто ограничен GIL, либо трудно масштабируется без него.

- **SIMD и низкоуровневые оптимизации:** Polars/Arrow хранит данные в форматах, удобных для SIMD, и активно применяет векторные инструкции [3]. Pandas, работая через NumPy, тоже выигрывает от SIMD, но часто упирается в другие узкие места (например, управление индексами, Python-объекты в колонках типа object). Polars минимизирует такие накладные расходы.

- **Формат данных (Arrow vs NumPy):** Arrow-формат экономнее по памяти и лучше для кеширования. Например, отсутствие Python-объектов для строк, битовые карты для null, колоночное выравнивание – все это дает Polars преимущество. Pandas же в некоторых случаях тратит в 5-10 раз больше памяти на аналогичные данные (особенно со строками) и чаще пролистывает лишние данные [7]. Больший объем данных = больше работа памяти, медленнее выполнение.

- **Ленивая модель вычислений:** Polars может откладывать исполнение цепочки операций и оптимизировать весь *pipeline* целиком. Это ведет к устранению лишних вычислений (pushdown фильтров, чтение только нужных столбцов и т.п.). Pandas выполняет шаги последовательно без глобальной оптимизации. В сложных задачах (несколько последовательных трансформаций) Polars может объединить их и сократить работу, тогда как Pandas сделает все шаг за шагом, включая ненужные [16]. Даже в eager-режиме Polars частично пользуется оптимизатором для отдельных запросов, тогда как Pandas ничего подобного не делает.

- **Продуманность алгоритмов:** Многие алгоритмы в Polars вдохновлены практиками из баз данных и систем big data. Например, Polars при группировке может автоматически выбирать алгоритм (сортировка vs хеш), оптимизировать порядок операций,

использовать стриминг для больших данных. Pandas же эволюционировал как инструмент для среднего объема данных и его алгоритмы не всегда оптимальны при выходе за RAM или при очень больших размерах.

Наконец, стоит упомянуть **Arrow-бекенд в Pandas 2.0**. Он направлен на улучшение использования памяти и, теоретически, производительности. С Arrow-поддержкой Pandas хранит данные как колонки Arrow (аналогично Polars) и может даже вызывать часть вычислений через PyArrow. Однако, на данный момент (2023-2024) эта функциональность еще сырая: не все операции поддержаны, возможны накладные расходы на переходы, а в тестах Pandas+PyArrow часто *медленнее* классического Pandas [11]. Таким образом, **Polars на сегодняшний день остается впереди** благодаря изначально цельной высокопроизводительной архитектуре.

Polars: жадное и ленивое исполнение

Одно из главных отличий Polars – поддержка двух режимов выполнения: **жадного (eager)**, как в Pandas, и **ленивого (lazy)**, как в SQL-движках или Spark. Разберем, в чем суть каждого подхода и как ленивость помогает оптимизации.

Различия подходов Eager vs Lazy

- **Жадное выполнение (Eager)** – каждое выражение или метод выполняется сразу же и возвращает результат. Этот подход привычен пользователям Pandas: вызвали `df.filter(...)` – тут же получили отфильтрованный DataFrame, вызвали `df.groupby().agg()` – сразу рассчитали агрегаты и т.д. Преимущество eager-режима – *простота отладки и интуитивность*: вы всегда оперируете конкретными данными на каждом шаге, можете посмотреть промежуточный результат. Недостаток – невозможность оптимизировать весь *конвейер* обработки, ведь вычисления уже произведены на каждом шаге. Например, если вы сначала отфильтровали данные, потом агрегировали, то при жадном подходе **весь набор данных читается, фильтруется, затем агрегируется** – даже если часть этих действий могла быть упрощена.

- **Ленивое выполнение (Lazy)** – при этом подходе вызовы методов не вычисляют результат сразу, а лишь строят *план вычисления*. Пользователь, по сути, описывает *что нужно сделать*, а не делает это моментально. В Polars при переходе в ленивый режим (через `df.lazy()` или чтение данных с помощью `pl.scan_csv`) вы получаете объект типа `LazyFrame`. Все последующие операции (фильтр, выбор колонок, группировка, объединение) над LazyFrame **не выполняются сразу**, а добавляются в *вычислительное дерево*. Реальное выполнение начнется только, когда вы явно вызовете `.collect()` (или `.fetch()` для части данных). Преимущество lazy-подхода – Polars имеет шанс **проанализировать весь запрос целиком и перестроить его оптимальным образом перед выполнением** [17]. Недостаток – более сложная логика (но Polars это скрывает, пользователь лишь видит, что результат появляется только после collect) и невозможность напрямую посмотреть промежуточные данные без дополнительных шагов.

Проще говоря, **жадный = сразу выполнить**, **ленивый = отложить и оптимизировать**. Оба режима доступны в Polars; по умолчанию методы вроде `pl.DataFrame.filter` работают жадно. Но если начать с `df.lazy()`, то можно построить цепочку лениво.

Примеры: ленивый vs жадный режим в Polars

Рассмотрим конкретный пример и сравним оба подхода. Задача: прочитать CSV-файл с данными, отфильтровать строки по условию, вычислить группировку и агрегат. Сначала сделаем это в жадном режиме (как Pandas), а затем лениво:

Жадное выполнение (Eager Polars):

```
import polars as pl

# 1. Чтение CSV (жадно, сразу читает весь файл в память)
df = pl.read_csv("data/iris.csv")  # допустим, файл Iris dataset

# 2. Фильтрация данных (результат - новый DataFrame с отфильтрованными
строками)
df_small = df.filter(pl.col("sepal_length") > 5)

# 3. Группировка и агрегирование (сразу выполняется на df_small)
df_agg = df_small.groupby("species").agg(pl.col("sepal_width").mean())

print(df_agg)  # печатаем результат
```

Этот код очень похож на Pandas-стиль: после каждого шага данные уже получены. Недостаток: **на шаге 1 читается весь CSV**, хотя мы потом интересуемся лишь частичной информацией. На шаге 2 мы фильтруем, но уже потратили время и память на все данные. Возможно, мы не используем и некоторых колонок (`petal_length`, `petal_width` не участвуют). Однако Polars выполнил все буквально: прочитал все колонки, затем отбросил лишнее.

Теперь то же самое, но с использованием **ленивого режима**:

```
# 1. Чтение CSV лениво (создаем LazyFrame, данные пока не читаются полностью)
q = pl.scan_csv("data/iris.csv")

# 2. Задаем последовательность преобразований лениво
q = q.filter(pl.col("sepal_length") > 5) \
     .groupby("species") \
     .agg(pl.col("sepal_width").mean())

# 3. Выполняем вычисления явно
result = q.collect()
print(result)
```

Обратите внимание: `pl.scan_csv` – ленивый аналог `read_csv`. Он не загружает данные немедленно, а лишь готовит *сканер*, способный читать файл покусочно. После него мы вызываем `.filter` и `.groupby().agg(...)` на `q` – эти вызовы мгновенно не пробегают по данным, а лишь регистрируют операции в плане. Переменная `q` – это `LazyFrame`, хранящий последовательность: "прочитать CSV -> отфильтровать -> агрегировать". Только при `q.collect()` Polars действительно выполняет все шаги.

Что здесь выигрывается? Полярный движок проанализирует план `q` перед выполнением и применит оптимизации:

- **Predicate pushdown (проталкивание фильтра):** Polars увидит фильтр `sepal_length > 5` и попытается применить его как можно раньше. В данном случае, при чтении CSV, Polars сможет пропускать строки, не удовлетворяющие условию, уже на этапе чтения. Проще говоря, вместо чтения 100% файла и потом отфильтровывания 50%, он может читать только нужные строки. Это экономит и время, и I/O.

- **Projection pushdown (чтение необходимых столбцов):** В запросе агрегации участвуют только колонки `sepal_length`, `sepal_width` и `species`. Значит Polars при разборе увидит, что **колонки** `petal_length` **и** `petal_width` **нигде не используются**, и не будет их даже считывать из файла [18] [19]. В жадном варианте мы читали все 5 колонок iris.csv, а в ленивом – лишь 3 нужные. Опять экономия памяти и времени.

- **Единое выполнение агрегата:** Вместо того чтобы хранить промежуточный `df_small`, Polars может слить шаги фильтрации и группировки. Он будет группировать данные "на лету", отфильтровывая их сразу при чтении. Таким образом, **нет лишних промежуточных структур**, всё делается в одном проходе (в память загружается только столько, сколько нужно для агрегата по группам).

Описанные оптимизации – частный случай. Polars optimizer умеет и другие вещи: устранять неиспользуемые вычисления, переупорядочивать фильтры и проекции, объединять последовательно примененные фильтры в один, переписывать некоторые выражения на более эффективные. В итоге **ленивое выполнение позволяет достичь максимальной производительности, автоматически сокращая объём работы** [16].

Оптимизации в ленивом режиме

Перечислим основные оптимизации, которые делает Polars при ленивом выполнении (многие взяты из реляционных СУБД):

- **Проталкивание фильтров (Predicate Pushdown):** Как было сказано, фильтры (`filter` / `where` условия) применяются как можно раньше, часто ещё на этапе чтения данных из источника. Это уменьшает объем обрабатываемых данных на следующих шагах [18].

- **Проталкивание проекций (Projection Pushdown):** Чтение только необходимых столбцов. Если вы выбрали 5 столбцов из набора с 50 колонками, Polars не станет читать все 50 с диска – он запросит у форматного читателя (CSV/Parquet) только нужные поля [18] [19]. В Pandas вы тоже можете указать `usecols` при чтении CSV вручную, но Polars делает это автоматически на основе дальнейшего плана запроса.

- **Агрегация при чтении:** Если вы делаете агрегат без промежуточных результатов (например, `df.filter(...).groupby(...).agg(sum)` как единый запрос), Polars может выполнить агрегирование во время одного прохода по исходным данным. Это экономит память – не надо хранить все отфильтрованные данные, а сразу считать промежуточные суммы/счетчики по группам.

- **Перестановка операций:** Polars может менять местами независимые операции, если это не влияет на результат, но ускоряет выполнение. Например, переместить вычисление

выраженного столбца (`with_columns`) до фильтра, если этот столбец не влияет на фильтр – либо наоборот, отложить дорогое вычисление до применения фильтра, если оно требуется только для прошедших данных. В StackOverflow-ответе приведен пример: Pandas код сначала вычислял `df['name'].str.upper()`, потом фильтровал строки. Polars lazy сначала применит фильтр, *потом* приведет к верхнему регистру – чтобы не тратить ресурсы на строки, которые отфильтруются [20] [21]. Оптимизатор решает это автоматически.

- **Удаление лишних вычислений:** Если в ходе запроса вы вычисляете какой-то столбец, а потом нигде не используете, Polars это заметит и просто не будет выполнять эти расчеты. В Pandas вы бы впустую тратили время.

- **Объединение операций:** Несколько последовательных фильтров объединяются в один (логическое И между условиями). Последовательные `select/with_columns` могут объединяться, и т.д. Цель – минимизировать число проходов по данным.

- **Оптимизация соединений (joins) и резких операций:** Polars может выбирать алгоритмы соединения (sort-merge join vs hash join) в зависимости от данных, применяет слияние сортировок, и другие приёмы из мира баз данных. Это выходит за рамки нашей темы, но отмечаем, что ленивый режим и здесь дает простор для оптимизации.

Результатом всех этих техник является то, что **ленивые запросы Polars часто выполняются намного быстрее и потребляют меньше памяти**, чем эквивалентный набор шагов в Pandas или в жадном режиме. Особенно заметно это на сложных конвейерах с множеством шагов.

Практическое применение ленивого исполнения (пошагово)

Чтобы использовать ленивое выполнение в Polars и проконтролировать оптимизации, можно следовать таким шагам:

1. **Создать ленивый источник данных.** Вместо прямого `pl.read_csv` используйте `pl.scan_csv` (для CSV) или `pl.scan_parquet` (для Parquet) – это создаст `LazyFrame`, не загружая данные сразу. Если данные уже в DataFrame, вызвать `df.lazy()` для перехода в ленивый режим.

2. **Построить цепочку преобразований на LazyFrame.** Применяйте методы фильтрации, выборки колонок, агрегирования, объединения как обычно, но на объекте `LazyFrame`. Например:

```
lazy_q = df_lazy.filter(pl.col("X") > 0) \
               .with_columns((pl.col("Y")**2).alias("Y2")) \
               .groupby("Category").agg(pl.col("Y2").sum())
```

   Здесь `lazy_q` еще не выполнен, а лишь содержит план: Filter -> WithColumns -> GroupBy.

3. **Просмотреть план оптимизации (опционально).** Используйте метод `lazy_q.explain()` для вывода оптимизированного плана запроса. Polars покажет, какие оптимизации применены. Например, вы можете увидеть вывод:

```
AGGREGATE [sum(Y2) by Category]
  FROM
  DATAFRAME SCAN
  PROJECT 2/5 COLUMNS
  SELECTION: [X > 0]
  WITH_COLUMNS: [Y2 = Y^2]
```

Это означает: Polars будет сканировать исходный DataFrame, при чтении сразу применит фильтр `X > 0` (Selection) и возьмет только колонки `X, Y` (2 из 5), затем вычислит `Y2`, затем агрегирует по Category [22] [23]. Таким образом, мы проверяем, что pushdown сработал (Selection, Project в плане). Если указать `lazy_q.explain(optimized=False)`, можно увидеть исходный (неоптимизированный) план для сравнения [24] [25]. Также Polars предоставляет метод `lazy_q.show_graph()` для визуализации плана (требуется GraphViz) [26].

4. **Выполнить запрос и получить результат.** Вызовите `result = lazy_q.collect()`. Polars выполнит все операции согласно оптимизированному плану и вернет финальный `DataFrame` (или может быть `LazyFrame.fetch()` для выборки части результатов). На этом шаге можно замерить время выполнения и сравнить с жадным подходом. В большинстве случаев вы увидите значительный выигрыш.

5. *(Опционально)* **Профилировать выполнение.** Polars имеет функцию `lazy_q.profile()` – она выполняет запрос и возвращает tuple (DataFrame, профилировочная информация). Либо можно установить переменную окружения `POLARS_VERBOSE=1` перед запуском, тогда Polars будет выводить на консоль информацию о каждом этапе выполнения, времени и использованной памяти. Это помогает понять, где узкие места даже внутри ленивого плана.

Следуя этим шагам, вы реализуете ленивое исполнение и наглядно убедитесь, как Polars оптимизирует обработку данных. **Ленивый режим особенно полезен при сложных вычислительных пайплайнах**: когда есть много последовательных трансформаций, джойнов, агрегатов. В простых случаях (одна операция) выигрыша может и не быть, тогда проще использовать eager. Но если вы заранее знаете последовательность этапов обработки – определенно стоит воспользоваться ленивым API Polars для максимальной эффективности.

Заключение: Мы рассмотрели теоретическую разницу SIMD vs SISD, практические примеры оптимизации в Python, а также подробно изучили три библиотеки: NumPy, Pandas, Polars. Polars, благодаря современному подходу (колоночный формат, многопоточность, ленивые вычисления, SIMD-оптимизации), демонстрирует выдающуюся производительность на больших данных, существенно опережая классический стек Pandas/NumPy в ряде сценариев. При этом Polars сохраняет удобство и знакомость API. Понимая внутреннее устройство этих инструментов, вы сможете писать более эффективный код для обработки данных, осознанно используя возможности SIMD и преимущества ленивых вычислений.

Источники и ссылки:

- Flynn's Taxonomy и архитектуры CPU [1] [3]
- Использование SIMD в NumPy и Polars [4] [5]
- Сравнение Pandas и Polars, производительность операций [11] [12] [27]

- Разъяснение ленивого выполнения в Polars [16] [22]

---

[1] Understanding Vectorization in NumPy and Pandas | by Mike Flanagan | Analytics Vidhya | Medium
https://medium.com/analytics-vidhya/understanding-vectorization-in-numpy-and-pandas-188b6ebc5398

[2] [3] Polars — I wrote one of the fastest DataFrame libraries
https://pola.rs/posts/i-wrote-one-of-the-fastest-dataframe-libraries/

[4] [6] Which operations in numpy uses SIMD? - Stack Overflow
https://stackoverflow.com/questions/71059081/which-operations-in-numpy-uses-simd

[5] polars - Rust
https://docs.rs/polars/latest/polars/

[7] [8] [9] [10] Coming from Pandas - Polars user guide
https://docs.pola.rs/user-guide/migration/pandas/

[11] [12] [13] [14] [15] Pandas 2.0 vs Polars: The Ultimate Battle | by Priyanshu Chaudhary | CueNex | Medium
https://medium.com/cuenex/pandas-2-0-vs-polars-the-ultimate-battle-a378eb75d6d1

[16] [17] [20] [21] python - What are the advantages of a polars LazyFrame over a Dataframe? - Stack Overflow
https://stackoverflow.com/questions/76612163/what-are-the-advantages-of-a-polars-lazyframe-over-a-dataframe

[18] [19] [22] [23] Lazy API - Polars user guide
https://docs.pola.rs/user-guide/concepts/lazy-api/

[24] [25] [26] How to Inspect and Optimize Query Plans in Python Polars
https://stuffbyyuki.com/how-to-inspect-and-optimize-query-plans-in-python-polars/

[27] High Performance Data Manipulation in Python: pandas 2.0 vs. polars | DataCamp
https://www.datacamp.com/tutorial/high-performance-data-manipulation-in-python-pandas2-vs-polars