# **1. Многопроцессорные архитектуры с общей и разделяемой памятью – специфика и сравнение:**

Многопроцессорные архитектуры с общей памятью (SMP, Symmetric Multiprocessing) представляют собой системы, в которых несколько процессоров имеют доступ к общей физической памяти. Каждый процессор может выполнять независимые вычисления, обращаясь к общей памяти для обмена данными.

Преимущества SMP-архитектур:

* Простота программирования: процессы могут обмениваться данными через общую память без необходимости явной передачи сообщений.
* Масштабируемость: возможность добавления дополнительных процессоров для увеличения вычислительной мощности.

Однако SMP-архитектуры также имеют свои ограничения:

* Ограниченная пропускная способность памяти: если несколько процессоров активно обращаются к общей памяти, возникают задержки из-за конкуренции за доступ к данным.
* Проблемы когерентности кэша: каждый процессор имеет собственный кэш, и изменения, внесенные одним процессором в общие данные, должны быть видимы всем остальным процессорам.

В отличие от SMP, многопроцессорные архитектуры с разделяемой памятью (NUMA, Non-Uniform Memory Access) имеют несколько локальных памятей, к которым имеют прямой доступ только определенные процессоры. Процессоры могут обмениваться данными только через явное копирование или передачу сообщений.

Преимущества NUMA-архитектур:

* Увеличенная пропускная способность памяти: каждый процессор имеет локальный доступ к своей памяти, что уменьшает конкуренцию за доступ к данным.
* Большая масштабируемость: возможность добавления большего количества процессоров с их локальной памятью.

Однако NUMA-архитектуры также имеют свои ограничения:

* Сложность программирования: необходимость явно управлять передачей данных между процессорами и их локальными памятями.
* Возможные проблемы с когерентностью кэша: изменения в локальной памяти одного процессора не всегда мгновенно видимы другим процессорам.

Сравнение SMP и NUMA архитектур зависит от конкретных требований и характеристик приложения. SMP-архитектуры обеспечивают простоту программирования и хорошую масштабируемость для приложений с интенсивным взаимодействием между процессорами. NUMA-архитектуры обеспечивают более высокую пропускную способность памяти и хорошую масштабируемость для приложений с ограниченным взаимодействием между процессорами.

# **2. Подходы к декомпозиции крупных вычислительных задач на подзадачи для параллельного исполнения:**

При параллельном исполнении крупных вычислительных задач важно разделить задачу на более мелкие подзадачи, которые могут быть выполнены параллельно. Это позволяет эффективно использовать ресурсы параллельной системы и ускорить выполнение задачи в целом. Существует несколько подходов к декомпозиции задач:

1. Разделение по данным (Data Partitioning): Задача разделяется на непересекающиеся наборы данных, и каждая подзадача работает с своим набором данных. Этот подход хорошо подходит для задач, в которых данные можно разделить на независимые части, такие как обработка массивов или таблиц.
2. Разделение по задачам (Task Partitioning): Задача разделяется на набор независимых подзадач, которые могут быть выполнены параллельно. Этот подход подходит для задач, в которых каждая подзадача выполняет определенную функцию или операцию над данными.
3. Гибридное разделение (Hybrid Partitioning): Используется комбинация разделения по данным и разделения по задачам для достижения наилучшей параллельной декомпозиции. Некоторые части задачи могут быть разделены по данным, а другие - по задачам, чтобы достичь оптимального баланса.
4. Декомпозиция по функциям (Functional Decomposition): В этом подходе задача разделяется на функциональные модули или компоненты, которые могут быть выполнены параллельно. Каждый модуль выполняет свою функцию, и данные передаются между модулями для обмена информацией. Например, в приложении для обработки изображений можно разделить задачу на модули для чтения изображения, обработки цветов, обнаружения объектов и сохранения результата. Эти модули могут работать параллельно, обрабатывая различные части изображения.

Выбор подхода к декомпозиции зависит от характера самой задачи, доступных ресурсов и требуемой производительности.

# **3. Модели параллельного программирования и их сочетаемость с архитектурами параллельных вычислительных систем:**

Модели параллельного программирования определяют способы описания и организации параллельных вычислений. Некоторые из наиболее распространенных моделей параллельного программирования включают:

1. Модель разделяемой памяти (Shared Memory Model):

- Соответствующие архитектуры: Многопроцессорные системы с общей памятью (SMP), многопоточные процессоры (Multithreaded Processors).

- Особенности: В этой модели несколько потоков или процессов имеют доступ к общей области памяти. Они могут обмениваться данными путем чтения и записи в разделяемые переменные. Примеры таких моделей включают OpenMP и POSIX threads (Pthreads).

2. Модель сообщений (Message Passing Model):

- Соответствующие архитектуры: Кластеры, многопроцессорные системы с распределенной памятью (MPP), гетерогенные системы.

- Особенности: В этой модели потоки выполнения (процессы) обмениваются сообщениями для передачи данных и синхронизации. Каждый поток (процесс) имеет свою собственную память, и обмен данными осуществляется явно через отправку и прием сообщений. Примеры таких моделей включают MPI (Message Passing Interface) и PVM (Parallel Virtual Machine).

3. Модель потоков данных (Dataflow Model):

- Соответствующие архитектуры: Динамические многопроцессорные системы, графические процессоры (GPUs), некоторые FPGA-системы.

- Особенности: В этой модели задачи представлены в виде узлов, которые исполняются, когда все их входные данные готовы. Каждый узел выполняется асинхронно и может обмениваться данными по мере необходимости. Эта модель позволяет максимально использовать параллелизм и ресурсы системы. Примеры таких моделей включают CUDA (Compute Unified Device Architecture) для программирования GPU и некоторые языки программирования для FPGA, такие как VHDL и Verilog.

4. Модель задач-агентов (Task-Based Model):

- Соответствующие архитектуры: Гетерогенные системы, многопоточные процессоры, системы с переменным набором ресурсов.

- Особенности: В этой модели задачи представлены в виде независимых агентов или задач, которые могут быть назначены и выполнены на различных ресурсах системы. Планировщик или рантайм-система распределяет задачи между доступными ресурсами и управляет их выполнением. Примеры таких моделей включают Intel TBB (Threading Building Blocks), Cilk+ и расширения OpenMP для задач.

Сочетаемость моделей параллельного программирования с архитектурами параллельных вычислительных систем зависит от поддержки их функциональности аппаратной платформой и программным окружением. Некоторые архитектуры могут предоставлять специальные механизмы для поддержки определенных моделей параллельного программирования, таких как распределение задач в SMP-архитектурах или механизмы коммуникации в NUMA-архитектурах. Важно выбирать модель параллельного программирования, которая наиболее эффективно сочетается с конкретной архитектурой для достижения наилучшей производительности и масштабируемости.

# **4. Профилирование реализации алгоритмов на Python, принципы решения задачи оптимизации производительности алгоритма:**

Профилирование реализации алгоритмов в Python позволяет определить узкие места и время выполнения различных частей кода, что помогает идентифицировать участки, требующие оптимизации. Процесс профилирования может включать следующие шаги:

1. Выбор инструмента профилирования: В Python существует несколько инструментов для профилирования, таких как cProfile, line\_profiler, memory\_profiler и другие. Выбор инструмента зависит от требуемой информации и уровня детализации, необходимых для анализа производительности.
2. Измерение времени выполнения: Запускайте код с профилировщиком и измерьте время выполнения различных частей алгоритма. Это поможет определить, где тратится больше всего времени.
3. Анализ профилировщика: Проанализируйте результаты профилирования и определите узкие места в коде. Это могут быть участки с большим количеством вызовов, долгим временем выполнения или большим потреблением памяти.
4. Идентификация оптимизационных возможностей: На основе результатов профилирования определите, какие части алгоритма можно оптимизировать. Это может быть устранение ненужных вычислений, улучшение алгоритма или использование более эффективных структур данных.
5. Реализация оптимизаций: Внесите изменения в код алгоритма, чтобы улучшить его производительность. Это может включать использование более эффективных алгоритмов, векторизацию операций, уменьшение количества обращений к памяти и другие оптимизации.
6. Повторное профилирование: После внесения изменений повторно запустите профилировщик, чтобы оценить эффективность оптимизаций и убедиться, что они привели к желаемому улучшению производительности.

Профилирование и оптимизация производительности алгоритма - это итеративный процесс, и важно продолжать его до достижения удовлетворительных результатов.

# **5. Проблема Global Interpreter Lock (GIL) в Python и способы обхода ее ограничений:**

GIL (Global Interpreter Lock) - это механизм, используемый в CPython (стандартная реализация Python), который ограничивает выполнение Python-кода только одним потоком в каждый конкретный момент времени. Это означает, что даже при использовании многопоточности в Python, выполнение не может происходить параллельно на нескольких ядрах процессора.

Однако, GIL не является проблемой для всех типов приложений. Он может оказаться значительным ограничением только для CPU-интенсивных задач, которые требуют параллельного выполнения на нескольких ядрах процессора. Для I/O-интенсивных задач, таких как работа с сетью или файлами, GIL не является проблемой, поскольку блокировка GIL освобождается при блокировке I/O-операций.

Существуют несколько способов обхода ограничений GIL:

1. Использование многопроцессорности (Multiprocessing): Вместо использования многопоточности, можно использовать многопроцессорность, запуская несколько процессов Python, каждый из которых имеет свой собственный интерпретатор и GIL. Межпроцессное взаимодействие может осуществляться с помощью механизмов, таких как очереди или сокеты.
2. Использование асинхронного программирования (Asynchronous Programming): Вместо использования многопоточности или многопроцессорности, можно использовать асинхронное программирование с использованием библиотеки asyncio. Асинхронное программирование позволяет обрабатывать I/O-операции без блокировки GIL, что может повысить производительность в некоторых случаях.
3. Использование потоков других языков (Using Threads from other languages): Если критическая часть кода является CPU-интенсивной, ее можно вынести в другой язык программирования, такой как C или C++, где отсутствует GIL. Python и этот язык можно связать вместе, чтобы обеспечить взаимодействие между ними.
4. Некоторые библиотеки, написанные на C или C++, могут обходить GIL, позволяя выполнять определенные операции без блокировки GIL. Например, библиотека NumPy выполняет многие операции над массивами данных без блокировки GIL, что позволяет достичь хорошей производительности для многих вычислительных задач
5. Использование других реализаций Python: Существуют другие реализации Python, такие как Jython, IronPython или PyPy, которые имеют отличную от CPython архитектуру и не используют GIL. В некоторых сценариях использование этих реализаций может быть полезным для обхода ограничений GIL.

# **6. Технологический стек Python для обработки и анализа данных, Python как glue language, специфика библиотеки NumPy и ее роль в экосистеме Python:**

Python обладает мощным технологическим стеком для обработки и анализа данных, благодаря разнообразным библиотекам и инструментам, которые составляют его экосистему. Вот основные компоненты технологического стека Python для обработки и анализа данных:

1. NumPy: NumPy (Numerical Python) - это фундаментальная библиотека для научных вычислений в Python. Она предоставляет мощные многомерные массивы данных, а также функции для операций линейной алгебры, преобразований массивов и математических операций. NumPy является одним из ключевых компонентов технологического стека Python и часто используется другими библиотеками для обработки данных.

2. Pandas: Pandas - это библиотека для манипулирования и анализа данных. Она предоставляет высокоуровневые структуры данных, такие как DataFrame, которые позволяют легко выполнять операции с данными, такие как фильтрация, группировка, агрегация, объединение и многое другое. Pandas облегчает работу с табличными данными и широко используется для предварительной обработки данных перед анализом.

3. Matplotlib и Seaborn: Matplotlib и Seaborn - это библиотеки для визуализации данных в Python. Они предоставляют множество функций и инструментов для создания различных типов графиков и диаграмм, что помогает визуализировать данные и исследовать их характеристики. Matplotlib предоставляет более низкоуровневый интерфейс, в то время как Seaborn предоставляет более высокоуровневые функции и стилизацию графиков.

4. SciPy: SciPy - это библиотека для научных и инженерных вычислений в Python. Она содержит набор модулей, включающих численные методы, оптимизацию, интерполяцию, обработку сигналов, статистику и многое другое. SciPy расширяет функциональность NumPy, предоставляя более высокоуровневые и специализированные функции для научных вычислений.

5. Scikit-learn: Scikit-learn - это библиотека машинного обучения в Python. Она предоставляет широкий спектр алгоритмов и инструментов для задач классификации, регрессии, кластеризации, обработки изображений, извлечения признаков и многое другое. Scikit-learn облегчает разработку и применение моделей машинного обучения на Python.

Python также широко используется как "glue language" (язык-склейка) в экосистеме данных. Это означает, что Python обеспечивает простой способ связи различных компонентов системы и библиотек. Он может использоваться для интеграции различных инструментов и библиотек данных, написанных на разных языках программирования, и для создания конвейеров обработки данных.

Библиотека NumPy играет ключевую роль в экосистеме Python для обработки и анализа данных. Ее основные особенности и роль включают:

- Многомерные массивы данных: NumPy предоставляет мощные структуры данных в виде многомерных массивов (ndarrays), которые позволяют эффективно хранить и манипулировать данными. Это особенно полезно для работы с числовыми данными и выполнения операций линейной алгебры.

- Быстрые математические операции: NumPy предлагает богатый набор функций для выполнения различных математических операций над массивами данных. Операции выполняются эффективно и оптимизированы для высокой производительности.

- Интеграция с другими библиотеками: Множество библиотек для обработки данных и научных вычислений в Python, таких как Pandas, SciPy и Scikit-learn, основаны на NumPy. Они используют массивы данных NumPy в качестве основного типа данных и взаимодействуют с ними.

- Быстрое выполнение кода: Благодаря своей эффективной реализации на языке C, NumPy обеспечивает быстрое выполнение вычислений над массивами данных. Он оптимизирован для использования многопоточности, векторизации и других техник, чтобы достичь высокой производительности.

# **7. Организация массивов в NumPy: хранение данных, создание массивов, принципы реализации операций с едиными исходными данными:**

В NumPy данные организуются в виде многомерных массивов, которые могут быть одномерными, двумерными, трехмерными и т.д. Массивы NumPy обеспечивают эффективное хранение и манипуляцию данными, что делает их основным объектом для работы с числовыми данными.

Для создания массивов в NumPy можно использовать различные методы, включая функции, такие как numpy.array, numpy.zeros, numpy.ones, numpy.arange и другие. Эти функции позволяют создавать массивы определенной формы и заполнять их значениями.

Операции с едиными исходными данными в NumPy выполняются покомпонентно. Это означает, что операции, такие как сложение, вычитание, умножение и деление, выполняются над каждым элементом массива независимо от других элементов. Например, если у вас есть два массива `a` и `b`, то операция сложения `a + b` будет выполняться покомпонентно, то есть каждый элемент `i` в `a` будет сложен с элементом `i` в `b`. Это работает для массивов любой формы, при условии, что их формы совпадают или совместимы для векторизованной операции. Это позволяет эффективно использовать векторные инструкции процессора и ускоряет выполнение операций над массивами.

# **8. Универсальные функции и применение функций по осям в NumPy:**

Универсальные функции (ufuncs) в NumPy - это функции, которые могут применяться к элементам массива поэлементно. NumPy предоставляет широкий набор универсальных функций для выполнения различных операций, таких как математические функции, логические функции, функции тригонометрии, функции сравнения и т.д.

Некоторые из наиболее часто используемых универсальных функций в NumPy включают:

- `numpy.add()`: Сложение элементов массивов.

- `numpy.subtract()`: Вычитание элементов массивов.

- `numpy.multiply()`: Умножение элементов массивов.

- `numpy.divide()`: Деление элементов массивов.

- `numpy.sin()`, `numpy.cos()`, `numpy.exp()`, `numpy.log()`: Тригонометрические и экспоненциальные функции.

- `numpy.logical\_and()`, `numpy.logical\_or()`, `numpy.logical\_not()`: Логические операции.

Пример использования универсальных функций:

*import numpy as np*

*a = np.array([1, 2, 3])*

*b = np.array([4, 5, 6])*

*c = np.add(a, b) # [5, 7, 9]*

Одной из особенностей универсальных функций в NumPy является их способность применяться по определенным осям массива. Ось - это измерение массива, и применение функций по осям позволяет выполнить операцию на всем измерении массива, свернуть его или агрегировать данные по определенной оси.

Например, с помощью функции numpy.sum можно вычислить сумму элементов массива по определенной оси. Это полезно, например, при работе с многомерными данными, где необходимо агрегировать или свернуть данные по определенным измерениям.

Некоторые из функций по осям в NumPy включают:

- `numpy.sum()`: Сумма элементов массива по указанной оси.

- `numpy.mean()`: Среднее значение элементов массива по указанной оси.

- `numpy.max()`, `numpy.min()`: Максимальное и минимальное значения элементов массива по указанной оси.

- `numpy.argmax()`, `numpy.argmin()`: Индексы максимальных и минимальных значений элементов массива по указанной оси.

Пример использования функций по осям:

*import numpy as np*

*arr = np.array([[1, 2, 3], [4, 5, 6]])*

*sum\_by\_columns = np.sum(arr, axis=0) # [5, 7, 9]*

*mean\_by\_rows = np.mean(arr, axis=1) # [2, 5]*

# **9. Принцип распространения значений при выполнении операций в NumPy:**

Принцип распространения значений (broadcasting) в NumPy позволяет выполнять операции между массивами различных форм без явного повторения данных. Он позволяет работать с массивами разных размерностей, путем автоматического преобразования и выравнивания массивов.

Общий алгоритм распространения значений в NumPy следующий:

1. Если массивы имеют разную размерность, меньший массив "расширяется" путем добавления новых осей с размерностью 1.
2. Если размерность массивов одинаковая, но их размеры по оси различаются, массив с размерностью 1 "расширяется" путем повторения значений по этой оси.
3. После расширения массивов они сравниваются поэлементно и выполняется операция.

Например, можно сложить массив размерности (3, 4) и одномерный массив размерности (4):

*import numpy as np*

*arr1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])*

*arr2 = np.array([1, 2, 3, 4])*

*result = arr1 + arr2*

В этом случае одномерный массив arr2 будет "расширен" до размерности (3, 4), повторяя значения по каждой оси. Затем будет выполнена операция сложения поэлементно.

# **10. Маскирование и прихотливое индексирование в NumPy:**

Маскирование (masking) и прихотливое индексирование (fancy indexing) - это два мощных инструмента в NumPy для выбора и манипуляции данными в массивах.

Маскирование позволяет создавать булевые массивы (маски), которые указывают, какие элементы должны быть выбраны или исключены из операций. Маски могут быть созданы на основе определенного условия или логического выражения. Затем маска может быть применена к массиву для выбора или исключения определенных элементов.

Например, можно создать маску для выбора элементов, удовлетворяющих определенному условию:

*import numpy as np*

*arr = np.array([1, 2, 3, 4, 5])*

*mask = arr > 2 # создание маски на основе условия*

*masked\_arr = arr[mask] # выбор элементов, удовлетворяющих маске*

*print(masked\_arr) # [3, 4, 5]*

Прихотливое индексирование позволяет создавать массивы, выбирая элементы из исходного массива на основе индексов, заданных в виде другого массива или списка индексов. Это дает возможность выбирать несколько элементов из массива в одной операции.

Например, можно выбрать определенные элементы массива на основе индексов:

*import numpy as np*

*arr = np.array([1, 2, 3, 4, 5])*

*indices = np.array([0, 2, 4]) # массив индексов*

*indexed\_arr = arr[indices] # выбор элементов по индексам*

*print(indexed\_arr) # [1, 3, 5]*

Маскирование и прихотливое индексирование предоставляют гибкость и возможность точного выбора нужных данных из массива.

# **11. Векторизация в numpy: ключевые параметры функции, примеры применения, использование обобщенной сигнатуры функции**

Векторизация в NumPy - это процесс применения операций к целым массивам, вместо выполнения операций поэлементно. Векторизованные операции выполняются намного быстрее, чем их эквиваленты, написанные с использованием циклов. Одной из ключевых особенностей векторизации в NumPy является использование универсальных функций (ufuncs).

Универсальная функция в NumPy - это функция, которая действует на массивы поэлементно, выполняя операцию над каждым элементом массива. Они позволяют выполнять поэлементные операции, такие как сложение, умножение, вычитание, деление и другие. У универсальных функций есть несколько ключевых параметров:

* out: Указывает массив, в который будет сохранен результат операции. Пример: `numpy.add(a, b, out=c)`, где `c` - массив, в который записываются результаты сложения `a` и `b`.
* where: Условие, которое определяет, где должна быть выполнена операция. Пример: `numpy.where(condition, x, y)`, где `condition` - условие, `x` и `y` - массивы, между которыми выбираются элементы в зависимости от условия.
* axis: Ось, по которой должна быть выполнена операция.

Примеры применения векторизации в NumPy:

*import numpy as np*

# Пример 1: сложение массивов

*arr1 = np.array([1, 2, 3])*

*arr2 = np.array([4, 5, 6])*

*result = arr1 + arr2*

# Пример 2: умножение массива на скаляр

*arr = np.array([1, 2, 3])*

*scalar = 2*

*result = arr \* scalar*

# Пример 3: применение тригонометрической функции

*arr = np.array([0, np.pi/2, np.pi])*

*result = np.sin(arr)*

Использование обобщенной сигнатуры функции позволяет применять векторизацию к пользовательским функциям. Обобщенная сигнатура функции позволяет применять операции к массивам любой формы, расширяя операцию поэлементно.

# **12. Numba: принципы работы, базовые примеры использования**

Numba - это компилятор функций Just-in-Time (JIT) для Python, который позволяет ускорять выполнение кода, особенно при работе с числовыми вычислениями и массивами. Numba преобразует функции Python в машинный код на лету, что позволяет достичь значительного увеличения производительности.

Принцип работы Numba включает следующие шаги:

1. Аннотация типов: Numba анализирует код функции и пытается определить типы аргументов и возвращаемого значения.
2. Генерация промежуточного представления: Numba создает промежуточное представление функции, которое используется для оптимизации и генерации машинного кода.
3. Оптимизация: Numba применяет различные оптимизации, такие как устранение лишних операций, слияние циклов и другие.
4. Генерация машинного кода: Numba генерирует машинный код для функции на основе промежуточного представления.

Примеры использования Numba:

*import numba*

*@numba.jit # Декоратор @jit для компиляции функции с помощью Numba*

*def my\_function(x):*

*result = 0*

*for i in range(x):*

*result += i*

*return result*

*# Вызов функции*

*result = my\_function(1000000)*

Numba также поддерживает параллельное выполнение с помощью декоратора @jit(parallel=True), который автоматически распараллеливает циклы в функции.

# **13. Организация Pandas DataFrame и организация индексации для DataFrame и Series**

Pandas DataFrame - это двумерная структура данных, представляющая собой таблицу с метками столбцов и строк. Он предоставляет удобные и мощные возможности для обработки, анализа и манипуляции табличных данных.

Организация DataFrame в Pandas основана на использовании двух основных структур данных: Series и NumPy ndarray. Series - это одномерная структура данных, представляющая собой столбец в DataFrame. ndarray - это многомерный массив, используемый для хранения данных в DataFrame.

Создание DataFrame:

*import pandas as pd*

# Создание DataFrame из словаря

*data = {'Name': ['John', 'Jane', 'Sam'], 'Age': [25, 30, 35], 'City': ['New York', 'London', 'Paris']}*

*df = pd.DataFrame(data)*

# Создание DataFrame из списка списков

*data = [['John', 25, 'New York'], ['Jane', 30, 'London'], ['Sam', 35, 'Paris']]*

*df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])*

Организация индексации в DataFrame и Series позволяет обращаться к отдельным элементам, столбцам или строкам в таблице данных. Индексация может быть выполнена с использованием различных методов:

* Целочисленная индексация: Обращение к элементам по их позиции в DataFrame или Series.

*df.iloc[0] # Возвращает первую строку DataFrame*

*df.iloc[:, 1] # Возвращает второй столбец DataFrame*

* Метка индексации: Обращение к элементам по их меткам (названиям) в индексе.

*df.loc[0, 'Name'] # Возвращает значение в первой строке и столбце 'Name'*

*df.loc[:, 'Age'] # Возвращает столбец 'Age'*

* Условная индексация: Выбор элементов, удовлетворяющих определенному условию.

*df[df['Age'] > 30] # Возвращает строки, в которых значение столбца 'Age' больше 30*

# **14. Применение универсальных функций и работа с пустыми значениями в Pandas**

В Pandas универсальные функции (universal functions, или ufuncs) представляют собой функции, которые могут быть применены к элементам данных в DataFrame или Series и выполняют операции поэлементно. Они позволяют эффективно выполнять операции на больших объемах данных без явного цикла.

Универсальные функции в Pandas могут быть применены к столбцам или рядам данных с помощью метода `.apply()` или встроенных методов, таких как `.mean()`, `.sum()`, `.max()`, `.min()`, и других.

Пример применения универсальной функции np.sqrt к столбцу DataFrame:

*import pandas as pd*

*import numpy as np*

*df = pd.DataFrame({'A': [1, 2, 3, 4, 5]})*

*df['B'] = df['A'].mean() # Применение встроенных универсальных функций к столбцу*

Работа с пустыми значениями (missing values) является важной задачей при работе с данными. В Pandas пустые значения могут быть представлены с помощью объекта NaN (Not a Number) из библиотеки NumPy. Pandas предоставляет функции и методы для обработки и управления пустыми значениями.

Некоторые полезные методы для работы с пустыми значениями в Pandas:

1. isnull() и notnull(): методы, которые возвращают логическое значение (True или False) для каждого элемента в DataFrame или Series, указывающее, является ли значение пустым или нет.

*df.isnull() # Возвращает DataFrame с логическими значениями, где True указывает на пустое значение*

*df.notnull() # Возвращает DataFrame с логическими значениями, где True указывает на не пустое значение*

1. dropna(): метод, который удаляет строки или столбцы, содержащие пустые значения.

*df.dropna() # Удаляет строки с пустыми значениями*

*df.dropna(axis=1) # Удаляет столбцы с пустыми значениями*

1. fillna(): метод, который заменяет пустые значения на указанное значение или выполняет заполнение пустых значений с использованием различных стратегий.

*df.fillna(value) # Заменяет пустые значения на указанное значение*

# **15. Объединение данных из нескольких Pandas DataFrame: общая логика и примеры:**

В Pandas существуют несколько способов объединения данных из нескольких DataFrame. Основные методы для объединения данных в Pandas включают concat(), merge() и join(). Вот общая логика и примеры каждого из этих методов:

1. concat(): Данный метод используется для объединения DataFrame вдоль определенной оси. Общая логика состоит в том, что данные просто прикрепляются друг к другу. Этот метод полезен, когда нужно объединить данные по вертикали или горизонтали.

Пример объединения нескольких DataFrame с помощью concat() по вертикали:

*import pandas as pd*

*df1 = pd.DataFrame( {'A': [1, 2, 3], 'B': [4, 5, 6]})*

*df2 = pd.DataFrame( {'A': [7, 8, 9], 'B': [10, 11, 12]})*

*result = pd.concat([df1, df2], axis=0) # Объединение по вертикали (по оси 0)*

1. merge(): Данный метод используется для объединения DataFrame на основе значений одного или нескольких столбцов. Общая логика состоит в том, что данные объединяются на основе совпадающих значений указанных столбцов.

Пример объединения нескольких DataFrame с помощью merge() по общему столбцу:

*import pandas as pd*

*df1 = pd.DataFrame( {'A': [1, 2, 3], 'B': [4, 5, 6]})*

*df2 = pd.DataFrame( {'A': [7, 8, 9], 'B': [10, 11, 12]})*

*result = pd.merge(df1, df2, on='A') # Объединение по столбцу 'A'*

1. join(): Данный метод используется для объединения DataFrame на основе индексов. Общая логика состоит в том, что данные объединяются на основе совпадающих индексов.

Пример объединения нескольких DataFrame с помощью join() по общему индексу:

*import pandas as pd*

*df1 = pd.DataFrame( {'A': [1, 2, 3], 'B': [4, 5, 6]})*

*df2 = pd.DataFrame( {'A': [7, 8, 9], 'B': [10, 11, 12]})*

*result = df1.join(df2) # Объединение по индексу*

В каждом из этих методов можно настроить параметры для определения типа объединения (например, внутреннее, внешнее, левое, правое) и управления обработкой пропущенных значений. Документация Pandas содержит подробные сведения о каждом из этих методов и их параметрах для более точного контроля над объединением данных.

# **16. Операция GroupBy в Pandas DataFrame и реализация в ней подхода "разбиение, применение и объединение":**

Операция GroupBy в Pandas DataFrame позволяет разбивать данные на группы в соответствии с заданными критериями, применять к каждой группе агрегирующую функцию и объединять результаты в новый DataFrame. Этот подход часто называют "разбиение, применение и объединение" (split-apply-combine) и представляет собой мощный инструмент для анализа данных.

Общая логика операции GroupBy состоит из следующих шагов:

1. Разбиение: Исходный DataFrame разбивается на группы на основе заданного критерия. Критерий может быть столбцом DataFrame, содержащим категориальные значения, или функцией, определяющей группировку на основе значений столбцов.
2. Применение: К каждой группе применяется агрегирующая функция или набор функций. Это может быть стандартная агрегирующая функция, такая как сумма, среднее или количество, или пользовательская функция, определенная пользователем.
3. Объединение: Результаты агрегации объединяются в новый DataFrame, в котором каждая группа представлена одной строкой, а результаты агрегации - значениями в соответствующих столбцах.

Пример использования операции GroupBy:

*import pandas as pd*

*df = pd.DataFrame({'key': ['A', 'B', 'A', 'B', 'A'], 'value': [1, 2, 3, 4, 5]})*

*# Группировка данных по ключу 'key' и вычисление суммы значений*

*grouped = df.groupby('key').sum()*

*mean\_values = df.groupby('key').mean()*

*count\_values = df.groupby('key').count()*

Операция GroupBy предоставляет также множество других функций для агрегации данных, таких как sum(), count(), min(), max() и др. Кроме того, можно использовать свои собственные пользовательские функции для агрегации данных.

Операция GroupBy в Pandas позволяет эффективно анализировать и обрабатывать данные, проводить групповые вычисления и извлекать статистическую информацию из DataFrame, основываясь на заданных критериях группировки.

# **17. Специфика текстовых и бинарных файлов, форматы файлов CSV и Pickle, представление данных в этих форматах и взаимодействие с ними в Python:**

Текстовые и бинарные файлы являются двумя основными типами файлов для хранения и представления данных в Python.

Текстовые файлы, такие как файлы CSV (Comma-Separated Values), представляют данные в виде текстовых строк, разделенных определенным разделителем (обычно запятой или точкой с запятой). Файлы CSV могут быть прочитаны и записаны с использованием стандартных библиотек Python, таких как csv или pandas.

Пример чтения данных из файла CSV с помощью библиотеки pandas:

*import pandas as pd*

*df = pd.read\_csv('data.csv') # Чтение файла CSV*

Бинарные файлы представляют данные в их сыром двоичном формате. Они могут содержать любые типы данных и обычно требуют специфической обработки для чтения и записи. Один из популярных форматов бинарных файлов в Python - формат Pickle, который позволяет сериализовать и десериализовать объекты Python.

Пример сериализации и десериализации объекта с использованием Pickle:

*import pickle*

*# Сериализация объекта*

*data = {'name': 'John', 'age': 30}*

*with open('data.pkl', 'wb') as file:*

*pickle.dump(data, file)*

*# Десериализация объекта*

*with open('data.pkl', 'rb') as file:*

*loaded\_data = pickle.load(file)*

# **18. Задача сериализации и десериализации, описание формата файла JSON и пример описания данных в этом формате и взаимодействия с ним в Python:**

Задача сериализации и десериализации данных состоит в сохранении объектов в виде последовательности байтов или текста, чтобы их можно было сохранить или передать по сети, а затем восстановить обратно в исходное состояние. Формат JSON (JavaScript Object Notation) является одним из наиболее популярных форматов для сериализации данных. Он обеспечивает простой и читаемый способ представления структурированных данных.

Формат файла JSON основан на двух основных типах данных: объектах и массивах. Объекты представляются в виде наборов пар "ключ-значение", где ключи являются строками, а значения могут быть строками, числами, логическими значениями, массивами, другими объектами или значениями null. Массивы представляют собой упорядоченные списки значений, которые могут быть любого типа данных JSON.

Пример описания данных в формате JSON:

{ "name": "John", "age": 30, "city": "New York", "pets": ["dog", "cat"]}

В Python для работы с данными в формате JSON используется модуль json. Он предоставляет функции для сериализации (кодирования) и десериализации (декодирования) данных.

Пример сериализации объекта в формат JSON:

*import json*

*data = { "name": "John", "age": 30, "city": "New York", "pets": ["dog", "cat"]}*

*json\_data = json.dumps(data)*

*print(json\_data)*

Вывод:

*{"name": "John", "age": 30, "city": "New York", "pets": ["dog", "cat"]}*

Пример десериализации данных из формата JSON:

*import json*

*json\_data = '{"name": "John", "age": 30, "city": "New York", "pets": ["dog", "cat"]}'*

*data = json.loads(json\_data)*

*print(data["name"]) # Выводит: John*

*print(data["pets"]) # Выводит: ['dog', 'cat']*

Таким образом, модуль json позволяет легко сериализовать объекты Python в формат JSON и обратно, обеспечивая удобное взаимодействие с данными в этом формате.

# **19. Формат XML и модель DOM: общая характеристика, пример описания данных в XML и DOM, работа с ними с помощью библиотеки BeautifulSoup**

Формат XML (Extensible Markup Language) - это язык разметки, используемый для структурирования данных в читаемой и расширяемой форме. Он представляет собой текстовый формат, который описывает иерархическую структуру данных с использованием тегов и атрибутов.

Модель DOM (Document Object Model) - это программный интерфейс, который представляет XML-документ в виде иерархической структуры объектов. DOM предоставляет методы и свойства для доступа, изменения и манипуляции элементами и атрибутами XML-документа. Он позволяет программно просматривать и модифицировать содержимое XML-документа.

Работа с XML и моделью DOM в Python с использованием библиотеки BeautifulSoup:

Библиотека BeautifulSoup предоставляет удобные методы для разбора XML-документов и работы с моделью DOM.

Таким образом, библиотека BeautifulSoup позволяет удобно работать с XML-документами и моделью DOM, извлекать данные, модифицировать их и выполнять другие операции на основе структуры XML-разметки.

# **20. Форматы файлов NPY и HDF - общая характеристика, пример взаимодействия с данными этих форматов в Python**

Форматы файлов NPY (NumPy) и HDF (Hierarchical Data Format) являются двумя популярными форматами для хранения и обмена данными в Python.

1. Формат NPY:
   * Характеристики: Формат NPY представляет собой бинарный формат хранения многомерных массивов данных в NumPy. Он сохраняет данные массивов, их форму, тип данных и другие метаданные.
   * Преимущества: NPY-файлы обладают простотой и эффективностью хранения и загрузки массивов данных. Они поддерживаются широким спектром инструментов и библиотек для научных вычислений в Python.

2. Формат HDF:

* Характеристики: Формат HDF является более гибким и расширяемым форматом для хранения и организации больших объемов данных. Он поддерживает иерархическую структуру данных, множество типов данных, метаданные и сжатие данных.
* Преимущества: HDF-файлы предоставляют богатые возможности для работы с данными, включая масштабирование, поиск, фильтрацию и манипуляции с различными объектами данных внутри файла.

Пример взаимодействия с данными в форматах NPY и HDF в Python:

import numpy as np

import h5py

# Сохранение массива в формате NPY

arr = np.array([1, 2, 3, 4, 5])

np.save('array.npy', arr)

# Сохранение данных в формате HDF

data = np.random.randn(100, 100)

with h5py.File('data.hdf', 'w') as f:

В этом примере массив сохраняется в формате NPY с помощью функции np.save(). Для формата HDF данные сохраняются с использованием библиотеки h5py.

Оба формата файлов - NPY и HDF - предоставляют удобные способы хранения и обмена данными в Python, и выбор между ними зависит от конкретных требований вашего проекта и предпочтений.

# **21. Взаимодействие с Excel из Python с помощью XLWings: принципы работы и примеры использования**

XLWings - это библиотека, которая позволяет взаимодействовать с Excel из Python, позволяя автоматизировать задачи работы с электронными таблицами, включая чтение, запись данных, манипуляции формулами, форматирование и многое другое.

Принципы работы с XLWings:

1. Создание подключения к Excel: с помощью функции xlwings.Book().
2. Работа с листами и ячейками: обращение к листам и ячейкам в Excel, используя синтаксис XLWings. Например, sheet = book.sheets['Sheet1'] - получение доступа к листу, value = sheet.range('A1').value - получение значения ячейки.
3. Чтение и запись данных: Используйте методы и свойства объектов ячеек для чтения и записи данных в Excel. Например, sheet.range('A1').value = 'Hello, XLWings!' - запись значения в ячейку.
4. Выполнение операций форматирования: Используйте методы и свойства объектов ячеек для форматирования данных в Excel. Например, sheet.range('A1').api.Font.Bold = True - установка жирного шрифта.

Пример использования XLWings:

*import xlwings as xw*

*# Создание подключения к Excel*

*book = xw.Book('Workbook.xlsx')*

*# Получение доступа к листу*

*sheet = book.sheets['Sheet1']*

*# Чтение значения ячейки*

*value = sheet.range('A1').value*

*print(value)*

*# Запись значения в ячейку*

*sheet.range('B1').value = 'Hello, XLWings!'*

В этом примере создается подключение к файлу Excel 'Workbook.xlsx', получается доступ к листу 'Sheet1' и читается значение ячейки 'A1'. Затем в ячейку 'B1' записывается новое значение. Наконец, сохраняется файл и закрывается подключение к Excel.

# **22. Основы работы с регулярными выражениями: базовый синтаксис, примеры использования модуля re в Python**

Регулярные выражения (регулярные выражения) - это последовательность символов, используемая для поиска и сопоставления шаблонов в тексте. В Python регулярные выражения поддерживаются с помощью модуля re.

Базовый синтаксис регулярных выражений:

* Метасимволы: Регулярные выражения содержат метасимволы, которые представляют особые символы или классы символов. Например, . - соответствует любому символу, \d - соответствует цифре и т.д.
* Квантификаторы: Квантификаторы определяют количество повторений предыдущего элемента в регулярном выражении. Например, \* - ноль или более повторений, + - одно или более повторений и т.д.
* Группировка: Группировка позволяет объединять части регулярного выражения в одну группу. Например, (abc) - группа с соответствием подстроке 'abc'.

Примеры использования модуля re в Python:

*import re*

*# Извлечение совпадений*

*pattern = r'\d+'*

*text = 'There are 5 apples and 10 oranges'*

*matches = re.findall(pattern, text)*

*print(matches) # ['5', '10']*

В этом примере используется модуль re для извлечения совпадений в тексте. Функция re.search() возвращает объект Match, который позволяет проверить наличие совпадения. Функция re.findall() возвращает список всех совпадений.

Это лишь базовые примеры использования регулярных выражений в Python. Регулярные выражения предоставляют богатый набор возможностей для работы с текстовыми данными и позволяют выполнять различные операции, такие как поиск, извлечение и замена подстрок.

# **23. Сегментация и токенизация текста на естественном языке, стемминг и лемматизация, примеры на Python:**

Сегментация и токенизация текста на естественном языке относятся к процессу разделения текста на отдельные слова или фразы, называемые токенами. Токенизация полезна для анализа и обработки текстовых данных.

В Python для сегментации и токенизации текста на естественном языке можно использовать различные библиотеки, такие как NLTK (Natural Language Toolkit) и SpaCy. Рассмотрим пример использования библиотеки NLTK:

*import nltk*

*from nltk.tokenize import sent\_tokenize, word\_tokenize*

*# Загрузка данных и разделение на предложения*

*text = "....."*

*sentences = sent\_tokenize(text)*

*tokens = [word\_tokenize(sentence) for sentence in sentences] # Разделение предложений на токены (слова)*

Стемминг и лемматизация относятся к процессу приведения слова к его базовой или нормальной форме. Это помогает объединить разные формы слова в единый корень, что полезно для поиска, классификации и анализа текстовых данных.

Пример использования стемминга и лемматизации с использованием библиотеки NLTK:

*from nltk.stem import PorterStemmer, WordNetLemmatizer*

*from nltk.corpus import wordnet*

*# Пример стемминга*

*stemmer = PorterStemmer()*

*word = "running"*

*stemmed\_word = stemmer.stem(word)*

*print(stemmed\_word) # run*

Стемминг (портирование) приводит слово к его основе путем удаления аффиксов. Лемматизация использует более сложные правила, чтобы привести слово к его нормальной форме, основываясь на его части речи.

Обратите внимание, что в случае лемматизации необходимо указать часть речи слова, чтобы получить корректную нормализацию.

# **24. Расстояние Левенштейна: определение, алгоритм эффективного поиска оптимального редакционного предписания, пример поиска на Python:**

Расстояние Левенштейна (также известное как редакционное расстояние) измеряет минимальное количество редакторских операций (вставка, удаление, замена символов), необходимых для преобразования одной строки в другую.

В Python можно использовать модуль difflib для вычисления расстояния Левенштейна. Рассмотрим пример:

*import difflib*

*str1 = "kitten"*

*str2 = "sitting"*

*# Вычисление расстояния Левенштейна*

*distance = difflib.ndiff(str1, str2)*

*edit\_ops = [op for op in distance if op[0] != ' ']*

*levenshtein\_distance = len(edit\_ops)*

*print(levenshtein\_distance) # 3*

В данном примере расстояние Левенштейна между строками "kitten" и "sitting" равно 3. Это означает, что для преобразования строки "kitten" в строку "sitting" необходимо выполнить три редакторские операции.

# **25. Векторное представление текста на естественном языке: общий алгоритм подходов TF; TF-IDF:**

Векторное представление текста на естественном языке является важной задачей в обработке текстовых данных. Два распространенных подхода к векторизации текста - это подход Term Frequency (TF) и подход Term Frequency-Inverse Document Frequency (TF-IDF).

Алгоритм подхода TF:

1. Разделить текст на отдельные слова (токены).
2. Подсчитать количество вхождений каждого слова в тексте.
3. Представить каждый текст в виде вектора, где каждый элемент вектора соответствует количеству вхождений слова в тексте.

Пример использования подхода TF на Python с использованием библиотеки sklearn:

Алгоритм подхода TF-IDF:

1. Разделить текст на отдельные слова (токены).
2. Подсчитать Term Frequency (TF), то есть количество вхождений каждого слова в тексте.
3. Вычислить Inverse Document Frequency (IDF) для каждого слова в корпусе текстов. IDF вычисляется как логарифм отношения общего количества документов к количеству документов, содержащих данное слово.
4. Умножить TF на IDF для каждого слова и представить каждый текст в виде вектора, где каждый элемент вектора соответствует значению TF-IDF для слова в тексте.

Пример использования подхода TF-IDF на Python с использованием библиотеки sklearn:

# **26. Модуль multiprocessing – назначение и основные возможности, API multiprocessing.Pool:**

Модуль multiprocessing в Python предоставляет возможности для выполнения параллельных вычислений, используя многопроцессорные архитектуры. Он позволяет создавать и управлять процессами, распределять задачи между ними и обмениваться данными между процессами.

Основные возможности модуля multiprocessing включают:

1. Создание процессов: Модуль позволяет создавать новые процессы с помощью класса Process. Каждый процесс имеет свою собственную память и исполняемый код.
2. Распределение задач: Модуль multiprocessing предоставляет класс Pool, который упрощает распределение задач между процессами. Pool предоставляет методы для выполнения функций или методов асинхронно и параллельно.
3. Обмен данными: Модуль multiprocessing предоставляет несколько механизмов для обмена данными между процессами. Один из таких механизмов - это очередь (Queue), которая позволяет процессам обмениваться объектами через очередь. Есть также другие механизмы, такие как разделяемая память (Value и Array) и блокировки (Lock, Rlock, Semaphore, Event, Condition).
4. Синхронизация: Модуль multiprocessing предоставляет механизмы синхронизации для предотвращения гонок данных и обеспечения согласованности доступа к общим ресурсам. Это включает в себя блокировки (Lock, Rlock), семафоры (Semaphore) и события (Event).

API multiprocessing.Pool предоставляет простой интерфейс для распределения задач между процессами в пуле. Он позволяет создать пул процессов с помощью класса Pool и использовать его для выполнения функций или методов асинхронно и параллельно. Вот несколько методов, которые предоставляет multiprocessing.Pool:

* apply(func, args): Выполняет функцию func с аргументами args в одном из процессов пула и возвращает результат выполнения.
* map(func, iterable): Применяет функцию func к каждому элементу из итерируемого объекта iterable и возвращает список результатов в том же порядке.
* map\_async(func, iterable): Асинхронно применяет функцию func к каждому элементу из итерируемого объекта iterable и возвращает объект AsyncResult, который может быть использован для получения результатов.

Это лишь некоторые из основных возможностей модуля multiprocessing и API multiprocessing.Pool. Модуль multiprocessing предоставляет мощные средства для выполнения параллельных вычислений в Python и управления многопроцессорными архитектурами.

# **27. Различия между потоками и процессами:**

* Процесс: Это экземпляр выполняющейся программы, обладающий собственным адресным пространством, памятью, стеком вызовов и ресурсами, такими как файлы и сокеты. Процессы работают независимо друг от друга и общаются через механизмы межпроцессного взаимодействия.
* Поток: Это легковесный исполнитель внутри процесса. В отличие от процессов, потоки разделяют общее адресное пространство и ресурсы с другими потоками в пределах одного процесса. Потоки позволяют выполнять множество задач параллельно внутри одного процесса.

Различие между различными планировщиками в Dask:

Dask предоставляет различные планировщики для распределенных вычислений:

* Local: Этот планировщик выполняет вычисления локально на одной машине. Он может использоваться для тестирования и отладки, а также для выполнения вычислений, которые не требуют распределения по нескольким узлам.
* ThreadPool: Этот планировщик использует потоки для распределения вычислений на нескольких ядрах одной машины. Он подходит для вычислений, которые не являются CPU-интенсивными и не блокируют GIL (глобальная блокировка интерпретатора Python).
* ProcessPool: Этот планировщик использует процессы для распределения вычислений на нескольких ядрах и/или нескольких машинах. Он подходит для CPU-интенсивных вычислений или для выполнения кода, который блокирует GIL.
* Distributed: Этот планировщик предназначен для распределенных вычислений на кластере из нескольких машин. Он позволяет масштабировать вычисления и управлять задачами на разных узлах кластера.

# **28. Граф зависимостей задач – суть структуры данных, ее построение и использование в Dask**

Граф зависимостей задач (Task Graph) является основной структурой данных в библиотеке Dask. Он представляет собой ориентированный ациклический граф, в котором вершины представляют собой задачи, а ребра определяют зависимости между задачами. Граф зависимостей задач используется для организации параллельных вычислений и управления выполнением задач.

В Dask граф зависимостей задач строится при выполнении вычислений над высокоуровневыми объектами, такими как Dask.Array, Dask.DataFrame или Dask.Delayed. Когда вы выполняете операции над этими объектами, Dask автоматически строит граф зависимостей задач, отображающий все промежуточные вычисления и зависимости между ними.

Процесс построения графа зависимостей задач в Dask может быть представлен следующим образом:

1. Создание задач, представляющих вычисления, например, выполнение функции или операции над данными.
2. Определение зависимостей между задачами, то есть указание, какие задачи должны быть выполнены перед другими задачами.
3. Построение графа, в котором каждая задача представлена вершиной, а ребра определяют зависимости между задачами.
4. Планирование выполнения задач, то есть определение порядка и способа выполнения задач с учетом их зависимостей и доступных ресурсов.

Граф зависимостей задач в Dask позволяет эффективно организовать параллельные вычисления и автоматически разбить их на мелкие задачи, которые могут выполняться независимо друг от друга. Это позволяет использовать распределенные вычислительные ресурсы и повышает производительность выполнения операций.

После построения графа зависимостей задач в Dask вы можете использовать различные методы для его выполнения, например, вызов функции compute() или использование методов объектов Dask.Array или Dask.DataFrame для вычисления результата.

Пример использования графа зависимостей задач в Dask:

*import dask*

*# Создание задач*

*task1 = dask.delayed(function1)(arg1)*

*task2 = dask.delayed(function2)(arg2)*

*task3 = dask.delayed(function3)(arg3)*

*# Определение зависимостей*

*task3 = task3 + task1*

*task4 = task2 \* task3*

*# Построение графа*

*graph = dask.visualize(task4)*

*# Выполнение графа*

*result = task4.compute()*

В этом примере мы создаем три задачи (task1, task2 и task3), определяем их зависимости и строим граф зависимостей задач graph. Затем мы вызываем метод compute() для выполнения графа и получения результата result.

Граф зависимостей задач является мощным инструментом в Dask, который позволяет эффективно организовывать и выполнять параллельные вычисления.

# **29. Dask.Array – структура данных, специфика реализации и применения, процедура создания**

Dask.Array - это структура данных в Dask, которая представляет собой распределенный многомерный массив, схожий с ndarray из NumPy. Dask.Array разбивает массив на блоки и распределяет их по вычислительным ресурсам для выполнения операций над массивом параллельно и эффективно.

Специфика реализации Dask.Array состоит в следующем:

* Массив разбивается на блоки меньшего размера, которые могут быть обработаны независимо друг от друга.
* Операции над Dask.Array генерируют граф зависимостей задач, который оптимизируется и выполняется параллельно с использованием планировщика Dask.
* Dask.Array обеспечивает ленивую оценку, что означает, что операции выполняются только при необходимости и возвращают отложенные объекты, представляющие результаты вычислений.
* Dask.Array поддерживает широкий набор операций, аналогичных операциям в NumPy, и позволяет манипулировать данными на распределенных вычислительных ресурсах.

Процедура создания Dask.Array включает в себя разделение массива на блоки и создание Dask.Array с использованием функций, таких как dask.array.from\_array() или dask.array.from\_delayed().

Пример создания:

*import numpy as np*

*import dask.array as da*

*arr = np.random.rand(1000)*

*darr = da.from\_array(arr, chunks=100) # Создание Dask.Array с указанием размера блоков (chunks)*

# **30. Dask.Array - поддерживаемые операции и отличия от NumPy ndarray:**

Dask.Array предоставляет множество операций, которые аналогичны операциям, поддерживаемым NumPy ndarray. Некоторые из поддерживаемых операций включают:

* Математические операции: +, -, \*, /, \*\*, sqrt, exp, log, и другие.
* Агрегирующие операции: sum, mean, max, min, std, var, и другие.
* Индексирование и срезы: [], slice(), take(), concatenate(), reshape(), и другие.

Отличия Dask.Array от NumPy ndarray:

* Dask.Array работает с данными, которые не помещаются в оперативную память, и может эффективно обрабатывать массивы, которые больше доступной памяти.
* Dask.Array выполняет операции лениво и автоматически разбивает массив на блоки, что позволяет эффективно распараллеливать вычисления и оптимизировать использование ресурсов.
* Dask.Array поддерживает вычисления на распределенных вычислительных ресурсах, что позволяет масштабировать вычисления на кластерах из нескольких машин.

# **31. Dask.Bag - структура данных, специфика реализации и применения, процедура создания Dask.Bag**

Dask.Bag является структурой данных в библиотеке Dask, предназначенной для работы с неструктурированными данными. Dask.Bag представляет собой коллекцию элементов, которые могут быть любого типа и не обязательно иметь одинаковую структуру. Это позволяет обрабатывать и анализировать разнородные данные, такие как текстовые файлы, JSON-объекты, записи логов и другие.

Основные особенности и применения Dask.Bag:

* Dask.Bag поддерживает ленивую вычислительную модель, что позволяет работать с большими объемами данных, не загружая их полностью в память. Вычисления выполняются по требованию при выполнении операций над данными.
* Dask.Bag поддерживает параллельную обработку данных. Он автоматически разбивает данные на части и распределяет их по доступным вычислительным ресурсам для параллельного выполнения операций.
* Dask.Bag предоставляет удобный API для мэппинга, фильтрации, преобразования, группировки и свертки данных.
* Dask.Bag является устойчивой к ошибкам структурой данных. В случае возникновения ошибок при выполнении операций, Dask.Bag сохраняет промежуточные результаты и позволяет продолжить выполнение с места возникновения ошибки.

Процедура создания Dask.Bag включает следующие шаги:

1. Импортируйте модуль dask.bag из библиотеки Dask.
2. Создайте источник данных, например, список, файловую систему или функцию-генератор.
3. Используйте функцию from\_sequence, from\_files, from\_delayed или другие функции из модуля dask.bag для создания объекта Dask.Bag на основе источника данных.

Пример создания Dask.Bag из списка:

*import dask.bag as db*

*data = [1, 2, 3, 4, 5]*

*bag = db.from\_sequence(data)*

В этом примере мы создаем список data, а затем используем функцию from\_sequence для создания Dask.Bag bag на основе этого списка.

# **32. Организация вычислений с помощью Map / Filter / Reduce: общий принцип и специфика параллельной реализации обработки данных в Dask.Bag**

В Dask.Bag можно применять операции Map, Filter и Reduce для обработки данных параллельно и лениво. Эти операции позволяют выполнять преобразования и анализ данных на элементарном уровне.

* Операция Map: Применяет заданную функцию к каждому элементу Dask.Bag и возвращает новый Dask.Bag с преобразованными элементами. Операция Map может выполняться параллельно для каждого элемента данных.

Пример применения операции Map:

*result = bag.map(lambda x: x \* 2)*

* Операция Filter: Отфильтровывает элементы Dask.Bag на основе заданного условия и возвращает новый Dask.Bag с отфильтрованными элементами. Операция Filter может выполняться параллельно для каждого элемента данных.

Пример применения операции Filter:

*result = bag.filter(lambda x: x % 2 == 0)*

* Операция Reduce: Сводит все элементы Dask.Bag к единственному значению с помощью заданной агрегирующей функции. Операция Reduce может выполняться параллельно для различных частей данных, а затем объединять результаты в единое значение.

Пример применения операции Reduce:

*result = bag.reduce(lambda x, y: x + y)*

# **33. API Dask.Bag - функции мэппинга, фильтрации и преобразования**

Dask.Bag предоставляет API с различными функциями для мэппинга, фильтрации и преобразования данных:

* map(func): Применяет функцию func к каждому элементу Dask.Bag и возвращает новый Dask.Bag с преобразованными элементами.
* filter(func): Отфильтровывает элементы Dask.Bag на основе функции func и возвращает новый Dask.Bag с отфильтрованными элементами.
* pluck(key): Извлекает значение с заданным ключом из каждого элемента Dask.Bag и возвращает новый Dask.Bag с извлеченными значениями.
* map\_partitions(func):Применяет функцию func к каждому разделу (части) данных Dask.Bag и возвращает новый Dask.Bag с преобразованными разделами данных.
* concat(): Объединяет несколько Dask.Bag в один Dask.Bag.
* frequencies(): Подсчитывает частоту каждого уникального значения в Dask.Bag.
* fold(func): Сводит все элементы Dask.Bag к единственному значению с помощью функции func.

Пример использования этих функций:

*result = bag.map(lambda x: x \* 2)*

*result = bag.filter(lambda x: x % 2 == 0)*

*result = bag.pluck('name')*

*result = bag.map\_partitions(lambda part: len(part))*

*result = db.concat([bag1, bag2, bag3])*

*result = bag.frequencies()*

*result = bag.fold(lambda acc, x: acc + x)*

# **34. API Dask.Bag - функции группировки и свертки**

API Dask.Bag также предоставляет функции для группировки и свертки данных в Dask.Bag. Ниже перечислены основные функции для группировки и свертки в Dask.Bag:

1. groupby(key): Группирует элементы Dask.Bag по заданному ключу и возвращает новый Dask.Bag, содержащий группы элементов.

Пример использования groupby:

*result = bag.groupby('category')*

1. fold(binop, combine=None, initial=no\_default): Производит свертку элементов Dask.Bag с использованием бинарной операции binop. Опционально может быть задана функция комбинирования combine и начальное значение initial.

Пример использования fold:

*result = bag.fold(lambda x, y: x + y, initial=0)*

1. frequencies(): Возвращает словарь, содержащий частоту каждого уникального элемента в Dask.Bag.

Пример использования frequencies:

*result = bag.frequencies()*

1. topk(k, key=None): Возвращает k наибольших элементов Dask.Bag. Может быть задан ключ сортировки key.

Пример использования topk:

*result = bag.topk(5, key=lambda x: x['value'])*

Эти функции позволяют группировать элементы Dask.Bag по определенным ключам и выполнять свертку данных для агрегации или анализа. Они предоставляют мощные инструменты для обработки и анализа больших объемов данных в Dask.Bag.