**Слайд 1 -** Введение

**Слайд 2 –** Dask и его сильные стороны

Dask - -это фреймворк для распределенных и параллельных вычислений на базе языка python. Он был создан в 2014 году малоизвестной коммерческой компанией. Позже эта компания превратилась Anaconda Инкорпорэйтед. Сам Dask вырос из проекта, который назывался Блейз и финансировался DARPA (Управления перспективных оборонно-исследовательских проектов министерства обороны США). С тех пор Dask вырос до проекта, которым пользуются в NASA, NVIDIA и многих других кампаниях. Команда Dask утверждает, что он дарит каждому пользователю три преимущества: простоту, мощь и веселье.

Простота связана с тем, что Dask -это библиотека python, которая очень близка по идеологии и синтаксису к Pandas и Numpy, и совместима с их экосистемами, а также с другими инструментами data science, такими как: scikit-learn, pytorch, xbost и другими.

Мощь Dask заключается в том, что он позволяет обрабатывать огромные массивы данных (до 100Gb на локальной машине, до 100Тб на кластере). Кроме того Dask обладает таким джентельменским набором, как параллельные вычисления, планировщики задач, оптимизация графа задач из коробки, поддержка различных источников данных (файлы, базы данных, облака, хранилища Hadoop, Hugging Face, web-ресурсы).

Ну и с третьим пунктиком пока не все понятно. Итак, чтобы проверить, насколько близки к правде были разработчики, когда рекламировали своё решение, давайте попробуем разобраться, что это такое и как это работает.

**Слайд 3 -** DaskBag

У Dask, есть 3 основные сущности(класса), которые являются скелетом всего решения. Первая - это Bag или сумка. Bag -это неупорядоченная коллекция с повторами, то есть это класс, который по своим свойствам объединяет списки и множества. Сумка - это самая простая коллекция в экосистеме Dask, которая нужны для распараллеливания вычислений на неструктурированных наборах данных.

При всей своей простоте они позволяют выполнять параллельные вычисления и при этом работать с огромными не помещающимися в память списками объектов. При этом Bags имеют свои ограничения, вызванные их природой, а именно они не изменяемые, это им досталось от списков, и работают медленнее, чем Nampy Array и DataFrame, что досталось им от множеств. Ведь множества, как и сумки не упорядочены и к ним невозможно обратиться по индексу. Поэтому Bags используют исключительно для того, чтобы загрузить данные из сырого json или другого списочного файла, отчистить их от лишних строк, а затем превращают уменьшенные данные в DataFrame или Array для дальнейших вычислений.

**Слайд 4 -** Dask Array

Следующий класс – Dask Array. Фактически, - это массив, который состоит из множества, массивов. Для простоты понимания, давайте называть их кусочками. Для того, чтобы Dask Array существовал, нужны следующие компоненты:

1. Первое - Метаданные, которые включают информацию о типах данных внутри кусочков, а также о типа данных самих кусочков.
2. Второе – псевдоним или имя, привязанное к каждому кусочку и определяющее, какие элементы графа задач относятся к этому кусочку, на слайде – это x
3. Третье - это кортежи, содержащие размеры кусочков по каждому измерению. Так, на рисунке справа мы видим, что наш Dask Array имеет 2 измерения размером 3 на 4 кусочка. Внутри каждой ячейки помещен свой кусочек. В данном случае все кусочки одинаковые и имеют размерность 5 на 8 элементов.

Основное назначение Dask Array - это матричные вычисления, поэтому, как правило, он используется в крупномасштабной визуализации, в геномике, в сложных вычисленных алгоритмах для оптимизации и статистики. Возможности, которые предоставляет Dask Array, коротко причислены на слайде, но в целом они полностью согласуется с тем назначением, которые озвучено ранее и теми принципами, на которых он строится.

Заботливые разработчики также сформировали для нас основные рекомендации по использованию Dask Array. Так, они утверждают, что даже при всей своей оптимизации, Dask Array все же медленнее некоторых других библиотек. Поэтому его стоит использовать только тогда, когда данные действительно большие.

Еще команда Dask напоминает о том, что для эффективной работы с Array следует обдуманно подходить к выбору размера кусочков. С одной стороны, надо брать кусочки максимально большие, для того, чтобы минимизировать издержки на процедуры считывания и записи. Но при этом настолько меленькие, чтобы они помещались в память (как правило это означает, что кусок должен быть не больше, чем 50% от свободной оперативки).

Так же разработчики обращают наше внимание на то, что не только Dask под капотом использует распараллеливание задач. И если в вашем коде будет метод, который тоже реализует свое распараллеливание, то число одновременно выполняемых потоков вырастет многократно, что может привести к непредсказуемым последствиям. Так что параллельте, но аккуратно.

И последнее, разработчики в паре с DaskArray рекомендуют использовать XArray, который добавляют удобства при работе со сложными данными так, как это описано на слайде. Надо сказать, что XArray был первый проектом, из которого затем вырос Dask. Так что разработчики знают, о чём говорят.

**Слайд 5 -** DaskDataFrame

И наконец мы переходим к главному герою сегодняшней презентации – к DaskDataFrame. DaskDataFrame – это цепочка Pandas DataFrame, получившаяся путем деления начального большого объекта данных на кусочки вдоль одного столбца. Это как колбаса, которую порезали на слайсы.

Для того, чтобы DaskDataFrame существовал, нужны следующие компоненты:

1. Первое – это шаблон пустого Pandas DataFrame, из которого будут собираться кусочки при помещении их в память
2. Второе - Метаданные, которые включают информацию о названиях колонок внутри кусочков, а также об их типах данных.
3. Третье – псевдоним или имя, привязанное к каждому кусочку и определяющее, какие элементы графа задач относятся к этому кусочку, на слайде – это x
4. Третье - это границы или в терминологии Dask - divisions, по которым поделены кусочки
5. И последний - это граф задач со специальным набором ключей, содержащих имя кусочка и его индекс.

Если вы создали себе DaskDataFrame, то сможете посмотреть на эти компоненты, обративших к его атрибутам. Например, на слайде мы видим общую информацию о DaskDataFrame из атрибута мета и типы данных, которые должны содержаться в его колонках, из атрибута dtypes. На примере другого DaskDataFrame мы видим, как можно вывести границы кусочков, обратившись к атрибуту divisions. В данном случае дата фрейм разделён по временному столбцу, на 5 кусочков, начало каждого из которых отражено в выведенном списке. К каждому кусочку можно обратиться отдельно, воспользовавшись атрибутов partitions.

Такая структура DaskDataFrame придумана для того, чтобы решить 2 основные задачи, которые нам уже знакомы: обработка больших объемов данных, которые не помещаются в память и оптимизация вычислительных затрат за счёт распараллеливания задач. И тут в игру вступает следующий наш герой - …. Граф задач. Давайте откроем завесу тайны и объясним, что это такое.

**Слайд 6 -** DAG

Граф задач Dask имеет форму ориентированного ациклического графа или DAGa, узлами которого являются функции python, а рёбрами объекты. Как только мы создаём Dask объект, для него автоматически создается DaskDag, первым этапам в котором является чтение и(или) интерпретация данных. Затем, когда мы применяем к Dask объекту любые методы все они автоматически записываются в DAG, но ни одна из них НЕ выполняется.

Дело в том, что Dask основан на ленивых вычислениях. Это значит, что все применённые к объекту Dask методы, начиная с чтения, выполняться только тогда, когда мы его настоятельно попросим, вызвав метод compute.

Исключительно после команды compute Dask начинает анализировать сформированный DAG и оптимизировать его (разбивать большие задачи на маленькие, если это допустимо, изменять связи между задачами и их порядок). И только потом вычисления выполняются.

Звучит подозрительно, не так ли? Если вы не доверяете Dask или хотите понять, что и как он будет делать, то у вас есть возможность визуализировать DAG до и после оптимизации.

Так, на экране, представлена последовательность трех операций:

1. Шаг 1 – создаётся единичная матрица размером 15 на 15 элементов,
2. Шаг 2 - матрица преобразуется в DaskArray для чего делиться на 9 кусочков, размером 5 на 5 элементов
3. Шаг 3 – DaskArray транспонируется
4. Шаг 4 - DaskArray складывается со своей транспонированной редакцией

DAG для такого простого алгоритма выглядит устрашающе (справа на слайде). Чтобы было легче, разработчики придумали для нас граф высокого уровня (на рисунке слева). В нем операции разделены на слои или шаги, которые легко описать словами. В каждом слое можно посмотреть важную для нас информацию: названия применяемых методов, их входы и выходы, типы данных.

**Слайд 7 –** DAG-оптимизация

Помимо этого, как мы помним, команда Dask заботливо реализовала для нас алгоритмы оптимизации DAGа. И мы даже можем посмотреть на оптимизированный DAG, применив метод visualize с параметром optimize\_gradh.

Так сейчас выведен неоптимизированный граф для описанного ранее алгоритма и …. Теперь оптимизированный. И, кажется, он стал намного скромнее.

**Слайд 8 –** Sheduller

А мы переходим к четвертой сущности, на которой строится экосистема Dask - это Sheduller или планировщик задач. Изначально в Dask был всего один тип планировщика - это Single-machine scheduler, которые управлял локальными задачами в пределах одного компьютера. Однако по мере развития, в нем появился ещё один планировщик - это Distributed scheduler, он предлагает больше функций, может работать как локально, так распределённо, но при этом требует чуть больше усилий для своей настройки. Помимо классификации, основанной на типе архитектуры, планировщики можно разделять по уровню абстракции ресурсов и количеству потоков, которые они обрабатывают одновременно.

**Слайд 9 –** Типы Sheduller

В таком контексте можно выделить 5 вариантов планировщиков.

Первый – Local Threads, который выполняет вычисления с помощью локального ThreadPoolExecutor. Он легкий, не требует настройки и отличается низкими task overhead или по-русски накладными расходами (около 50 мкс на задачу). А еще он не тратит время на передачу данных между задачами так как все операции происходят в рамках одного процесса. Супер вариант! Надо брать. Но, из-за глобальной блокировки интерпретатора (GIL) он позволяет достичь значительного ускорения только при работе с НЕ питоновскими операциями, к которым, в частности, относятся работа в массивах NumPy, кадрах данных Pandas или использовании любого другого проекта на базе C/C++/Cython.

Второй - Local Processes, который выполняет вычисления с помощью локального ProcessPoolExecutor. Он легок в использовании, не требует настройки и способен обойти проблемы с GIL и обеспечить параллелизм даже в вычислениях, в которых доминирует чистый код Python. Это происходит за счет того, что задачи и все их зависимости передаются отдельным локальным процессам, выполняются, а затем их результат отправляется обратно в основной процесс. Однако перемещение данных в удаленные процессы и обратно может привести к снижению производительности, особенно если это большие данные. Local Processes — отличный выбор, когда рабочие процессы относительно линейны и поэтому не требуют значительной передачи данных между задачами.

Третий - Single Thread, который выполняет все вычисления в локальном потоке без какого-либо параллелизма. Это особенно ценно для отладки и профилирования, которые усложняются при использовании потоков или процессов.

Четвертый - Dask Distributed(local), который можно настроить в кластере или запустить на одном компьютере. Не смотря на имя, этот планировщик часто используют локально, потому что он обеспечивает доступ к асинхронному API, в частности к футуринам, и предоставляет диагностическую панель. А еще он обрабатывает процессы более сложно и поэтому может быть более эффективным, чем Local Processes. Кроме того, его всегда можно запустить на кластере и сделать распределенным.

**Слайд 10 –** Советы

Как и любым другим инструментом, Dask желательно пользоваться грамотно, только тогда ваши вычисления будут быстрыми и эффективными. Давайте же пробежимся по основным советам новичку.

Первое - Используйте любые инструменты BigData только тогда, когда данные не помещаются в память. Ведь это еще один уровень абстракции, а сложное враг простого.

Второе - Загружайте данные через Dask, уменьшайте их объем и переходите к Pandas или Numpy.

Третье – не расслабляйтесь и не забывайте о тех техниках оптимизации, которые характерны для Pandas, а именно: откажитесь от apply и transform, используйте матричные вычисления, не злоупотребляйте циклами.

Четвертое – помните Dask – не Pandas. Да, синтаксис, у них практически одинаковый, но при этом абсолютно разные подходы к вычислениям. Главное отличие - Dask ленивый и выполняет операции в самом конце. Помните – все, что вы в него вписали он не забудет и выполнит столько раз, сколько вписали. В связи с этим крайне не рекомендуется выполнять трансформацию данных непосредственно в DaskDataFrame, это может привести к непредсказуемым последствиям.

Четвертое – не забывайте задавать индекс, но делайте это аккуратно. Например, если у вас в данных есть столбец с датой, то иногда будет разумно использовать его как индекс. Тогда Dask сможет разделить данные на partitions, к примеру, по годам. И когда потребуется проводить агрегацию внутри года он точно будет знать, данные из какого кусочка ему взять. Это ускорит и оптимизирует вычисления. А ещё когда вы устанавливаете индекс и точно знаете, что этот столбец у вас отсортирован, расскажите об этом Dask, иначе ему придётся проверить сортировку во всех кусочках, и только потом окончательно определится с разбиением.

Пятое – в самом начале выберите себе какую-то тактику (то есть индекс) и придерживайтесь её. Знайте, - операция установки индекса и его замены очень вычислительно затратное мероприятие и им лучше не злоупотреблять.

Шестое – используйте кусочки оптимального размера – не слишком большого, чтобы не уйти в out of memory, и не слишком маленькие, – чтобы снизить накладные расходы.

Седьмое – объединяйте правильно. Join может выполняться, как очень быстро, так и очень медленно, быстро он будет вычисляться:

1. тогда, когда вы будете объединять DaskDataFrame с PandasDataFrame, потому что фактически это дозапись фрейма в последний кусочек
2. тогда, когда вы будете объединять два DaskDataFrame, но внутри одного кусочка, то есть кусочки не придётся перетасовывать
3. тогда, когда вы будете объединять DaskDataFrame по индексу

Ну и соответственно медленно, она будет происходить в тех случаях, когда Dask придётся часто перетасовывать данные.

**Слайд 11 –** Советы

Если все это обобщить получиться следующий алгоритм действий:

1. загружаем данные в Dask
2. исследуем данные с помощью Dask, то есть смотрим, какие там есть столбцы, какие у них типы
3. очищаем данные с помощью Dask на самом наивном уровне, например удаляем ненужные столбцы или пустые значения
4. отбираем данные, которые связаны с нужными нам сущностями, например с нужными нам хостами
5. рассчитываем статистики и фичи, проводим агрегации, но результат записываем не в начальный DataFrame, а в новый, которые потом переводим в Pandas.

Если это необходимо и нужно обогатить начальный DataFrame какими-то данными, то мы рассчитываем эти данные отдельно, а потом присоединяем к начальному фрэйму.

**Блокнот**

Всё это, конечно, супер здорово, но, как говорится, лучше 1 раз увидеть, чем 1000 раз услышать. Поэтому приглашаю посмотреть на блокнот.

Мы используем стандартный набор библиотек и функцию дисплея мемори, которая показывает, сколько у нас доступно памяти на компьютере и сколько памяти занято. Также мы используем вот такой набор данных из проекта eCar, которые преобразованы в разные форматы**.**