МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

Технологии хранения в системах кибербезопасности

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень	бакалавриат
	(бакалавриат, магистратура, специалитет)
Форма обучения	очная
	(очная, очно-заочная, заочная)
Направление(-я) подготовки	10.05.04 Информационно-аналитические системы безопасности
	(код(-ы) и наименование(-я))
Институт	Кибербезопасности и цифровых технологий (ИКБ)
	(полное и краткое наименование)
Кафедра	КБ-2 «Прикладные информационные технологии»
	(полное и краткое наименование кафедры, реализующей дисциплину (модуль))
Лектор	к.т.н., Селин Андрей Александрович
	(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года 2024/2025

(учебный год цифрами)

Проверено и согласовано «____» _____2024 г. _____ А.А. Бакаев

(подпись директора Института/Филиала с расшифровкой)

Москва 2024 г.

Технологии хранения в системах кибербезопасности

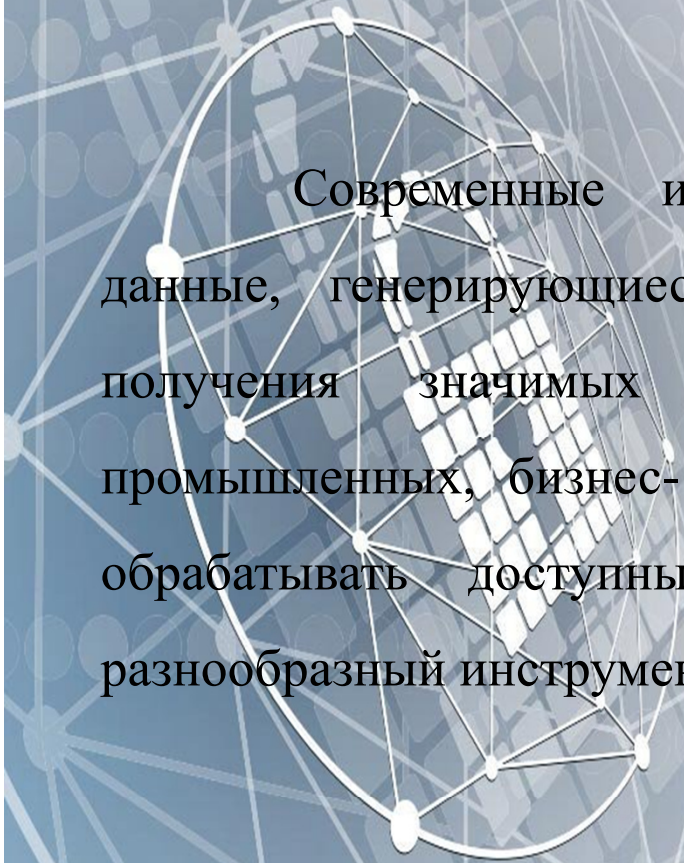
2024 год

Лекция 14.
Apache Spark обработка данных

Учебные вопросы лекции:

1. Основные этапы обработки данных
2. Управление памятью в Apache Spark

Введение



Современные информационные системы ориентированы на данные, генерирующиеся в огромных количествах ежедневно. Для получения значимых результатов в современных реалиях в промышленных, бизнес- или научных задачах необходимо эффективно обрабатывать доступные большие данные (Big Data), используя разнообразный инструментарий.

Основные этапы обработки данных

Запуск вычислений (и конструирование результатов или новых RDD) возможен для отдельных партиций – таким образом получают возможность эффективно работать операции *head, take* и *sample*. А также возможно частичное повторение вычислений, при котором потерянные партиции в уже материализованных RDD могут быть восстановлены независимо.

внутреннего представления и партиций с данными в партиции Spark);
формирование объектов задач и размещение их на соответствующих уровню локальности исполнителях;

- применение преобразований, определенных пользователем, над данными, что также в процессе требует управления размещением данных в оперативной памяти и на диске;
- применение служебных преобразований, результатами которых являются служебные файлы с данными, такие как shuffle-файлы;
- передача данных между узлами согласно формированию новых партиций – т.е. shuffle-операции. Такие операции могут происходить как в случае mapreduce – подобных вычислений, так и слияния двух датасетов или репартицирования исходного датасета;
- персистинг данных;
- выгрузка данных во внешнее хранилище.

Загрузка данных из внешнего хранилища

Как будет происходить загрузка данных, а также сколько партиций будет создано в процессе нее, определяется адаптером конкретного хранилища.

Как правило, типичными задачами такого адаптера являются:

- выполнение служебных запросов для определения количества партиций и получения всей необходимой метаинформации (например, схемы таблиц или идентификация схемы с помощью сэмплирования json-документов);
- определение местоположения партиций (что особенно актуально, если кластер хранилища и вычислительный кластер являются одним целым);
- подготовка и оптимизация запросов, специфичных для хранилища, для выборки данных из него, включая передачу необходимых параметров для фильтрации на стороне хранилища (если хранилище позволяет это);
- трансфер данных из источника и их запись во внутреннее представление, которое сможет использовать для вычислений Apache Spark.

При чтении данных из хранилища сначала будет выполнена операция подсчета, что тоже может потребовать достаточно интенсивных вычислений. Особенное внимание тут следует уделять при работе с источниками, где схема данных не известна наперед: json-файлы в hdfs, данные в Mongodb и т.п. Загрузка данных также является операцией вычислений над данными и происходит при материализации датасетов.

Изменение размещения данных и количества партиций

В процессе вычислений может возникнуть потребность в изменении размещения данных и / или количества партиций в датасете.

Например, после загрузки данных из файлов в hdfs нам необходимо, чтобы все данные, принадлежащие одному и тому же пользователю, попали на один узел, так как все дальнейшие операции будут происходить только в рамках одного пользователя.

Для достижения такого эффекта, можно воспользоваться функций *repartition* с указанием количества партиций или конкретного partitioner'а. В Spark по умолчанию доступно два основных partitioner'а, позволяющих добиться этой цели:

- HashPartitioner – размещает запись в партиции в соответствии с хэшем от ключа этой записи (актуально для key-value RDD).
- RangePartitioner – размещает запись в партиции в соответствии с диапазоном, в который попадает ключ данной записи (актуально для key-value RDD). Такой partitioner может быть полезен, например, в ситуации когда нам необходимо агрегировать данные по продажам за отдельные недели – тут RangePartitioner может помочь с размещением всех данных, относящихся к конкретной неделе только в одной партиции, тем самым уберая необходимость в реальной сетевой передачи данных.

Изменение размещения данных и количества партиций

В случае если RDD не назначен partitioner, распределение записей в партиции происходит равномерно. В случае DataFrame API репартицирование может быть применено к нескольким колонкам. Также существует возможность дописать свой собственный partitioner, например, для ситуации, когда нам имеет смысл разбивать данные и по идентификатору клиента, и по интервалу совершения операции – таким образом все данные одного пользователя за одну неделю окажутся в одной партиции. При этом, если пользователь имеет очень много активности, его обработку можно будет распараллелить (так как партиция – это минимальная единица последовательной обработки).

Размещать таким образом данные можно не только для одного датасета, а сразу для нескольких, например, имеющих один и тот же первичный ключ. В случае если нам понадобится проводить над ними операцию join, за счет одинаковых partitioner’ов и соответственно одинаковому расположению партиций с теми же ключами на узлах, получится избежать сетевой передачи в операции shuffle (при условии одинакового количества партиций в обоих RDD).

Как происходит вычисление над данными в Spark

Непосредственно запуском вычислений над данными в Spark управляет DAGScheduler, находящийся в драйвере приложения и создаваемых вместе с объектом SparkContext. DAGScheduler отвечает за:

- создание выполняемого графа приложения;
- генерацию задач, назначение и их рассылку на экзекьютеры, а также за мониторинг хода их выполнения и предоставлению пользователю этой информации пользователю;
- отслеживание событий отказа и принятие решения о перезапуске или остановке вычислений, а также об игнорировании определенных исполнителей.

Выполняемый граф приложений отличается от графа преобразования данных из-за оптимизаций, применяемых DAGscheduler.

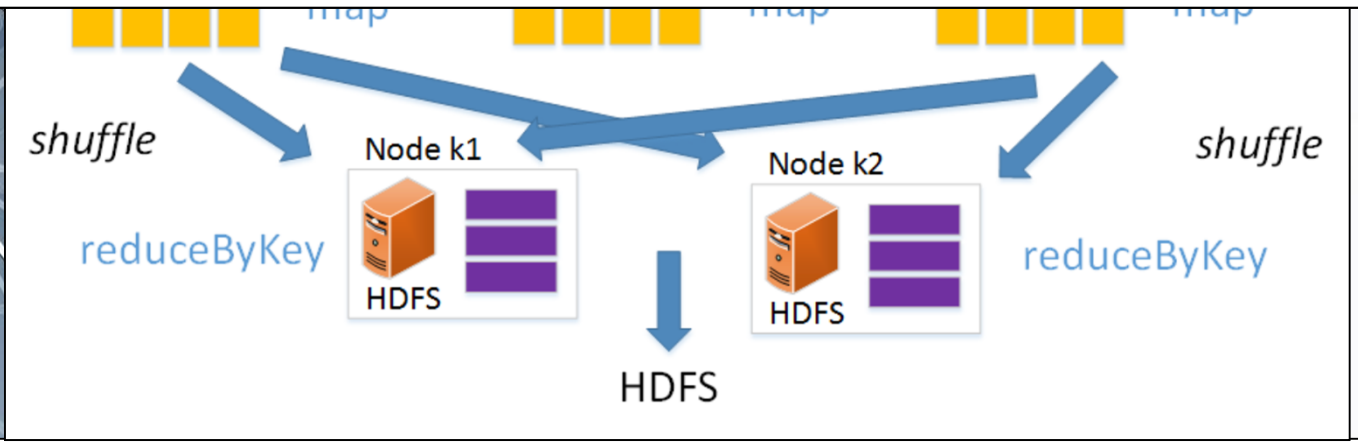
Материализованная форма графа состоит из стадий (stage). Стадия представляет собой последовательность RDD, связанных narrow зависимостями, функции которых объединены в одну единственную функцию обработки.

Результатом такой стадии является либо сетевой обмен данными между executor'ами и соответственно узлами кластера – т.е. операция shuffle, которая будет рассмотрена более подробно позднее – либо сохранение результатов во внешнее хранилище, либо, в определенных случаях, возвращение данных в driver приложения.

Как происходит вычисление над данными в Spark

Narrow зависимости – это зависимости, возникающие при операциях, в которых для получения партиции дочернего датасета нужна только одна партиция родительского датасета. Примерами таких операций служат map, filter, flatMap, mapPartitions – в них одна родительская партиция превращается в одну дочернюю.

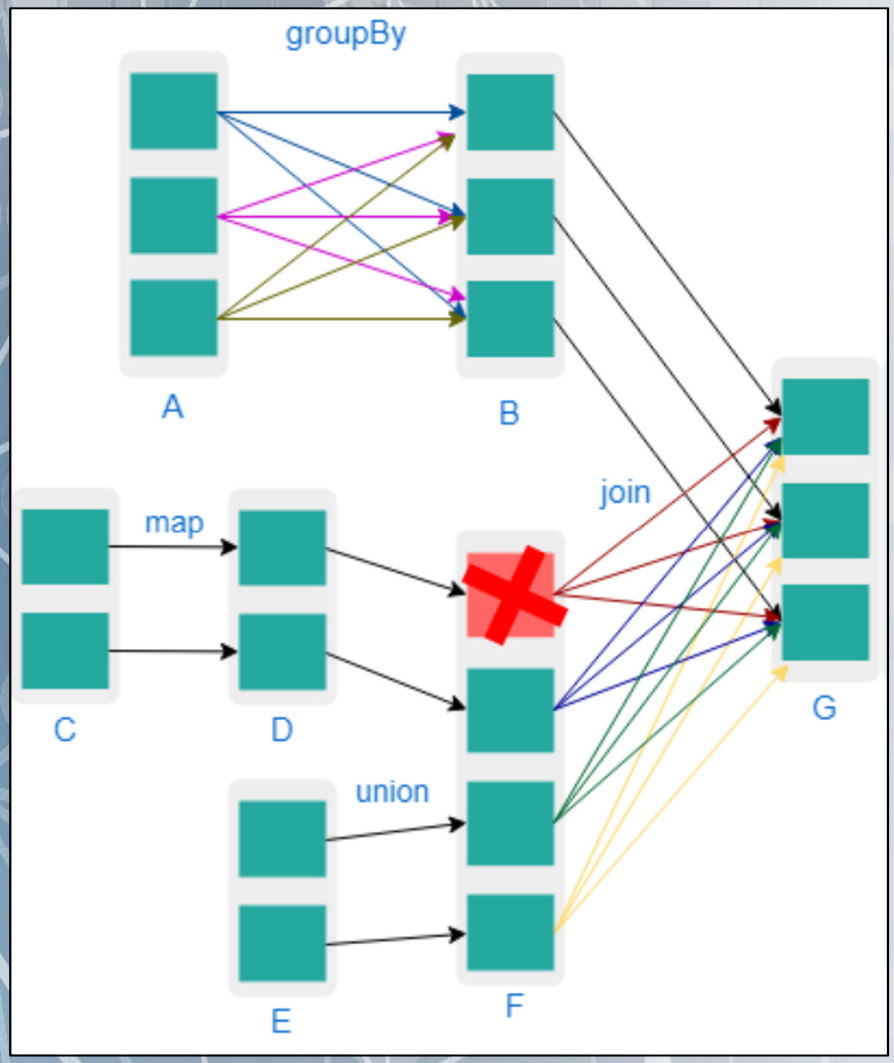
Wide зависимости – это зависимости, возникающие при операциях, в которых для получения партиции дочернего датасета нужны несколько или все партиции родительского датасета. Примерами таких операций служат groupByKey, reduceByKey, join, repartition. В примере, данном выше, как раз и используется такая операция для того, чтобы сгруппировать одни и те же слова для подсчета, но это может потребовать сетевой передачи данных, так как одни и те же слова могут встречаться в данных, лежащих на разных узлах.



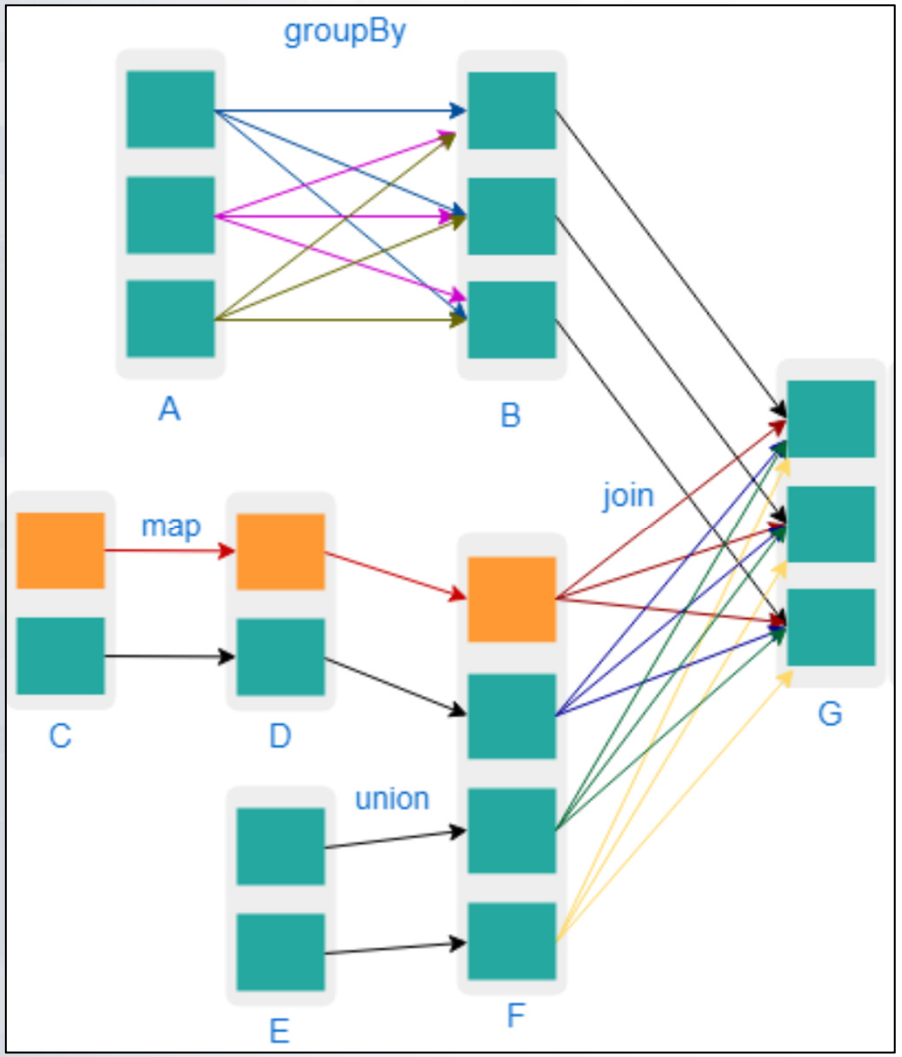
Объединение функций обработки по narrow зависимостям позволяет не генерировать промежуточных наборов данных и таким образом избежать ненужных накладных расходов на запись на диск данных, их сериализацию / десериализацию.

Как происходит вычисление над данными в Spark

Восстановление утраченных партиций по narrow зависимостям за счет частичного пересчета:



Потеря партиции в RDD F



Восстановление партиции в RDD F за счет пересчета родителей данной партиции

Как происходит вычисление над данными в Spark

Narrow зависимости не только более эффективны с точки зрения производительности, так как не предполагают дорогой сетевой передачи данных (а также подготовки к ней, включающей работу с диском и сериализацию), но и более надежны в ситуациях утраты части данных, так как в этом случае Spark позволяет пересчитать только утраченные партиции из неутраченных родительских партиций (в случае, если какой-то промежуточный RDD был закэширован) или даже партиций, получаемых из внешнего хранилища в самом начале обработки.

В случае же wide зависимостей одна дочерняя партиция может зависить от всех родительских, что и происходит в случае reduceByKey предыдущего примера. В таком случае для восстановления нескольких утраченных партиций дочернего RDD придется восстанавливать все партиции родительских RDD, если они тоже утрачены, что может быть очень затратно. Чтобы избежать таких проблем, в случае wide зависимостей рекомендуется использовать checkpointing.

Как происходит вычисление над данными в Spark

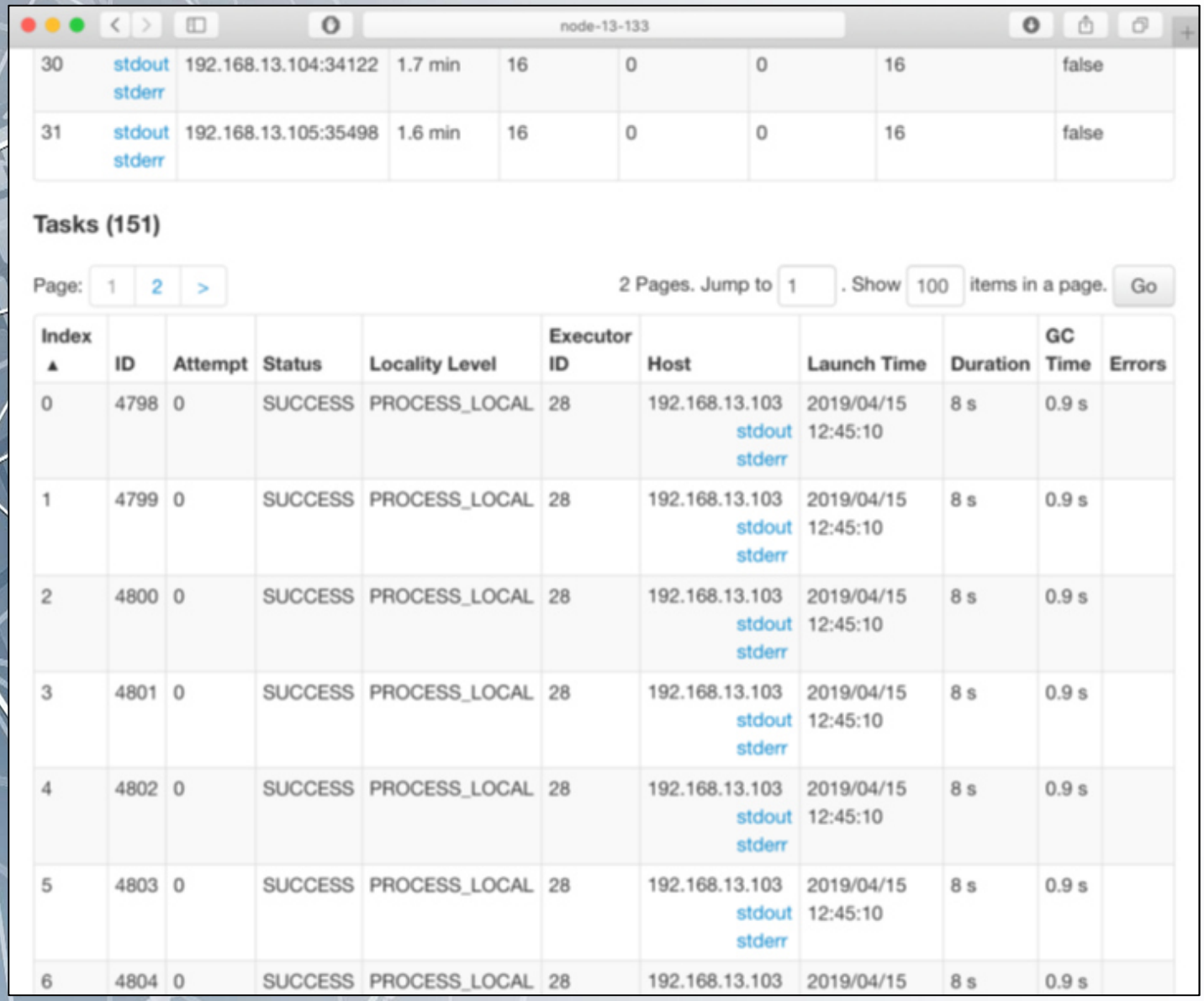
Стадия состоит из задач (task), каждая из которых генерируется на партицию в датасете.

Задачи затем рассылаются по исполнителям для их выполнения. Каждая из задач при этом содержит:

- сериализованную функцию (может состоять из последовательного набора заданных пользователем преобразований или сервисных преобразований) – на стороне исполнителя она будет десериализованна и ей будет предоставлен итератор к коллекции данных при выполнении;
- информацию о партиции, над данными которой необходимо выполнить вычисления;
- порядковые идентификаторы самой задачи и текущей попытки.

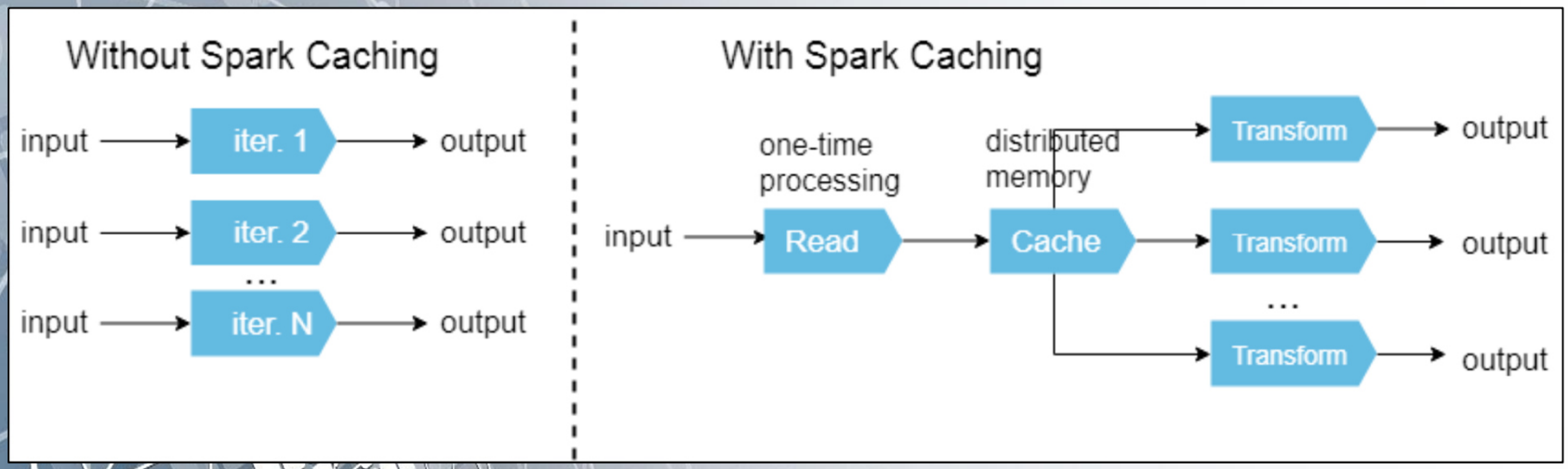
Задача также может содержать необходимые операции (согласно тому, что реализовано в адаптере к хранилищу) по загрузке данных из хранилища. В этом случае итератор не будет содержать данных, а задача сама соединится с хранилищем и загрузит данные. Кроме хранилища, источником данных для задач могут являться кэшированные данные, данные shuffle, загруженные с удаленной машины и данные чекпоинтов.

Задачи и их статусы в пользовательском интерфейсе приложения Spark



Ветвление и итеративные вычисления

Ветвление вычислений с помощью кэширования:

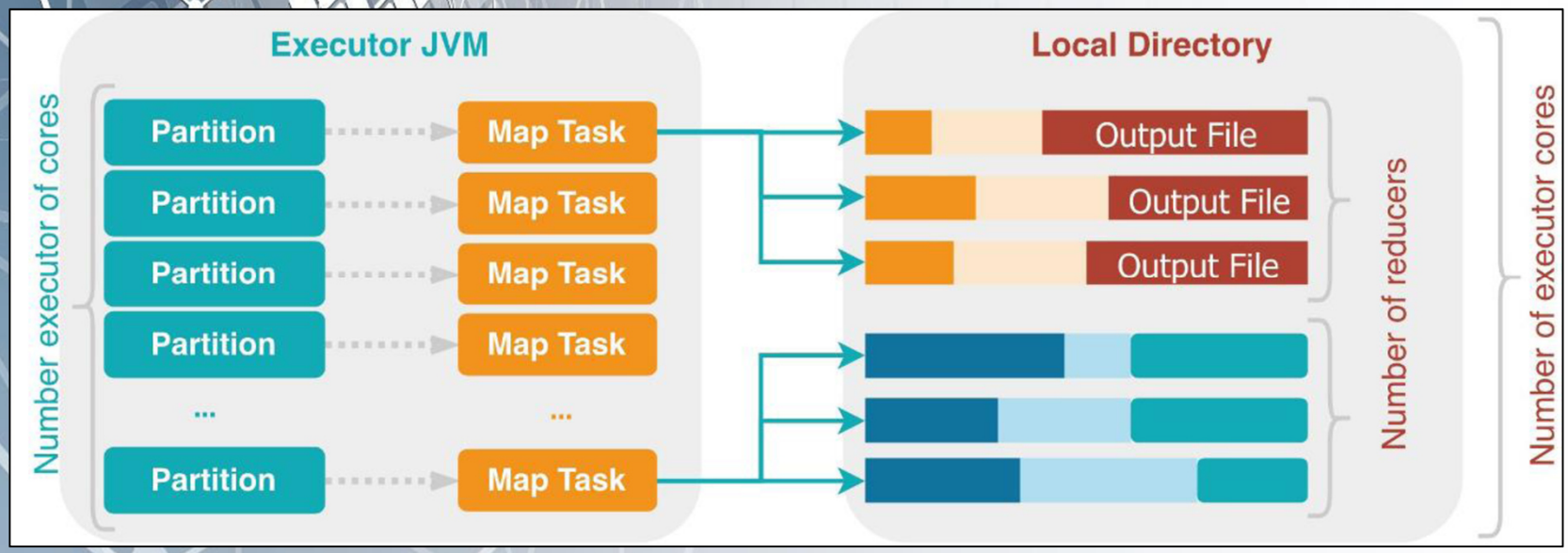


Возможность выполнять итеративные вычисления и обработку данных, повторно использующую те же самые данные, без дорогостоящей записи промежуточных результатов во внешнее хранилище – накладные расходы на сериализацию / десериализацию, запись на диск и чтение с диска, а также репликацию – позволяет существенно ускорить процесс выполнения. Повторное использование возможно за счет механизма кэширования (caching / persisting), имеющегося в Spark. Кэширование осуществляется с помощью вызова специальных трансформаций cache() или persists () (в последнюю нужно передавать уровень, на котором будут сохраняться данные: память; диск; одновременно память и диск; уровни с сериализацией).

Механизм Shuffle

Процесс shuffling является одним из фундаментальных, т. к. позволяет изменять размещение данных на узлах, являясь основой операций объединения (join) и группировки (group by, reduce) данных.

На вход shuffle поступает RDD с *n* партициями, а выходом будет RDD с *m* партициями. В силу исторических причин мы будем называть входной RDD map-стороной, а выходной – reduce-стороной. В Spark количество выходных партиций не зависит напрямую от количества уникальных ключей в конкретном датасете, а определяется параметром spark.sql.shuffle.partitions, который задается в настройках SparkContext. Каждая запись из партиций первого RDD будет оценена по некоторому ключу, и относительно него будет назначена партиция из выходного RDD, в которую эта запись попадет. Таким образом, все записи с одинаковыми ключами попадут в одну и ту же партицию выходного RDD.



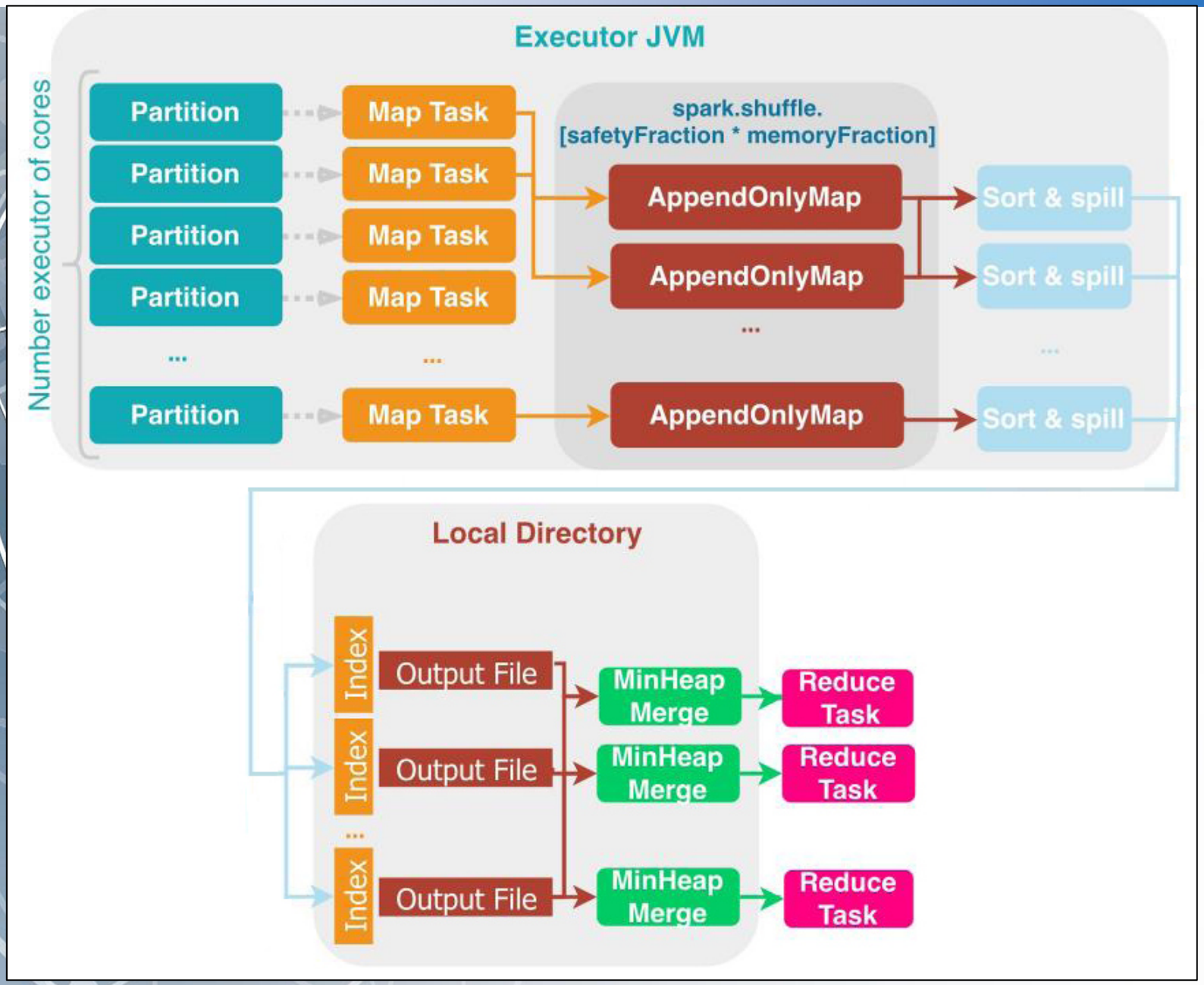
Механизм Shuffle

Для реализации shuffle необходима подготовка данных на map-стороне – группировка входных записей по их ключам, чтобы их затем можно было отослать на узел с нужной партицией.

Существует два основных способа, как это можно сделать. Первый способ (hash shuffle) подразумевает использование отдельного набора файлов для каждой map-задачи. Его оптимизированная версия, используемая в текущих версиях Spark, подразумевает переиспользование набора файлов для каждого из слотов.

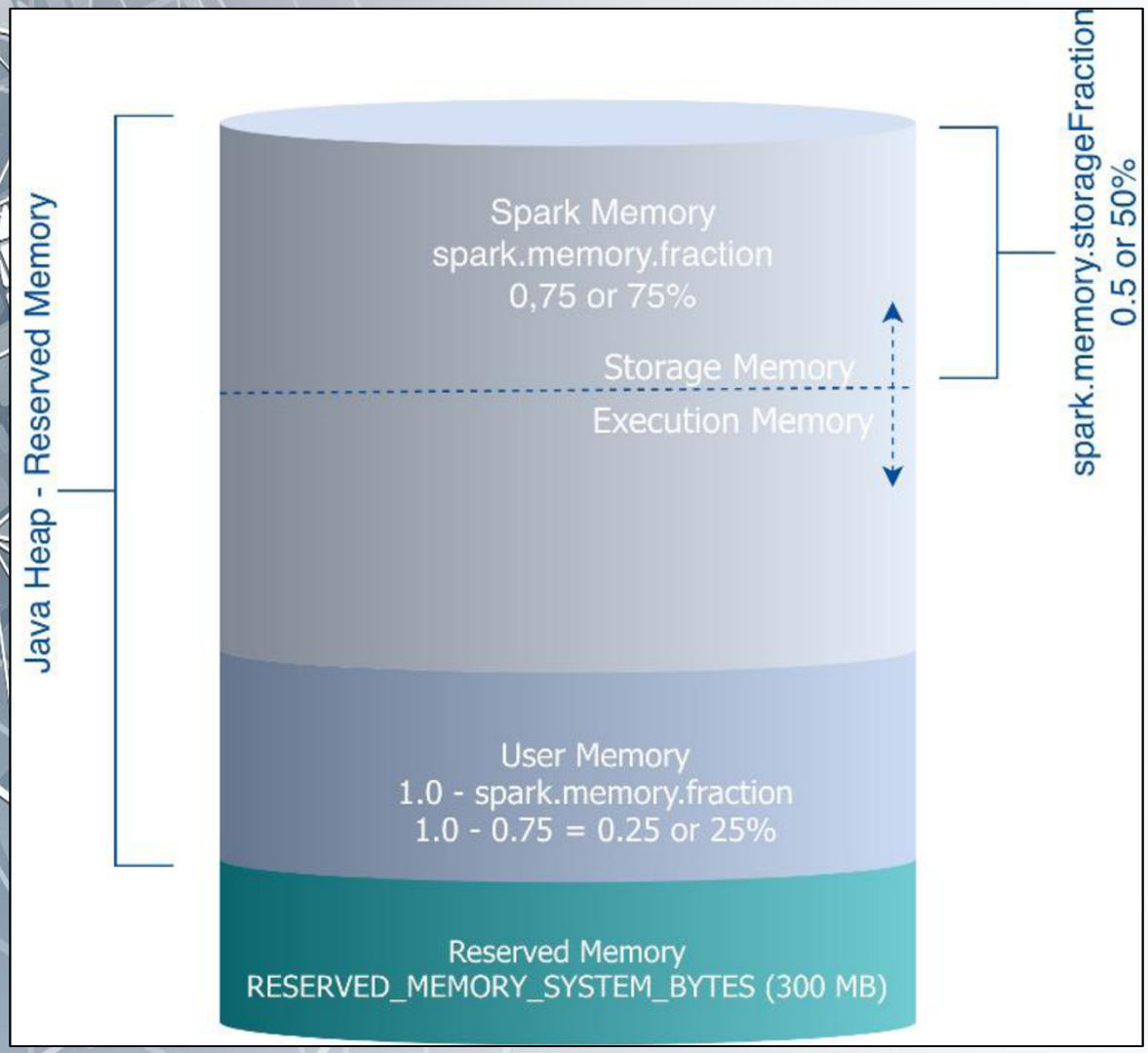
Sort Shuffle используют другую логику обработки. Hash shuffle на выходе производит по одному отдельному файлу для каждой выходной партиции и, как следствие, для каждого из «редьюсеров». С помощью же sort shuffle возможно сократить их количество: выходные файлы упорядочены по id «редьюсера» и содержат индекс в самом начале, состоящий из указателей позиций (offsets) на начало блока записей с конкретным id. Это позволяет легко получить блок данных, относящихся к «редьюсеру x», просто используя информацию о положении связанного блока данных в файле. Для небольшого количества «редьюсеров» очевидно, что хеширование отдельных файлов будет работать быстрее, чем сортировка, поэтому сортировка в случайном порядке имеет план «отката»: когда количество «редьюсеров» меньше, чем «spark.shuffle. sort.bypassMergeThreshold» (по умолчанию 200), используется hash shuffle.

Sort Shuffle



Управление памятью в Apache Spark

Модель памяти Spark предполагает наличие 3 основных областей памяти: **Reserved Memory, User Memory, Spark Memory.**



Управление памятью в Apache Spark

Зарезервированная память (Reserved Memory). Это память, зарезервированная системой, и ее размер жестко закодирован. Начиная с версии Spark 1.6.0, его значение составляет 300 МБ, что означает, что эти 300 МБ RAM не участвуют в вычислениях размера области памяти Spark, и его размер нельзя изменить каким-либо образом без перекомпиляции Spark или установки spark.testing.reservedMemory, что не рекомендуется, так как это параметр тестирования, не предназначенный для использования во время реальных вычислений.

Пользовательская память (User Memory). Это пул памяти, который остается после выделения Spark Memory, и может быть использован для хранения структуры данных пользователя, которые создаются в процессе вычислений с помощью трансформаций RDD и функций, задаваемых определяемых пользователем (user-defined function, UDF).

Пример: можно переписать агрегацию Spark, используя хеш-таблицу, и использовать трансформацию mapPartitions для запуска этой агрегации, которая будет потреблять пользовательскую память. В Spark 1.6.0 размер этого пула памяти можно рассчитать следующим образом:

(Java Heap–Reserved Memory) × (1 – spark.memory.fraction)

Параметр *spark.memory.fraction* по умолчанию равен 0,25.

Управление памятью в Apache Spark

Память Spark (Spark Memory). Это пул памяти, управляемый самим Spark. Его размер можно рассчитать следующим образом:

(Java Heap – Reserved Memory) × spark.memory.fraction

Например, если размер Java Heap в 4 ГБ, то этот пул будет иметь размер 2847 МБ. Весь этот пул разделен на 2 области - память хранения и память выполнения (Storage Memory and Execution Memory), и граница между ними задается параметром spark.memory.storageFraction, который по умолчанию равен 0.5. Преимущество этой новой схемы управления памятью состоит в том, что эта граница не является статической, и в случае нехватки памяти граница будет перемещаться, то есть одна область будет расти за счет заимствования пространства у другой.

Память хранения (Storage memory). Этот пул используется как для хранения кэшированных данных Spark, так и для временного развертывания сериализованных данных. Также все Broadcast переменные хранятся там как кешированные блоки с уровнем персистентности MEMORY_AND_DISK по умолчанию.

Память исполнения (Execution Memory). Этот пул используется для хранения объектов, необходимых во время выполнения задач Spark. Например, он используется для хранения промежуточного буфера на map этапе, а также для хранения хеш-таблицы для этапа агрегации по хэшу. Этот пул также поддерживает размещение данных на диске, если недостаточно памяти.

DataFrame API и Spark SQL. Датафреймы

RDD API, трансформации и действия которого были рассмотрены ранее, является достаточно низкоуровневым и используется в основном только для реализации некоторых алгоритмов машинного обучения, а также для поддержки унаследованных программ обработки данных (legacy code). Основным API на данный момент является DataFrame API, его основные отличия:

- Представление датасета и его трансформация осуществляется с помощью специальных объектов, называемых датафреймами. Датафрейм – это таблица со столбцами, а не просто набор записей, как это было в случае с RDD.
- Датафреймы с их схемами (метаинформация в виде описания колонок и их типов), доступными на каждом шаге обработки, позволяют реализовать структурированную обработку данных (structured processing). В этом случае ядро Spark имеет доступ к информации не только о типе всей записи, но и о каждом конкретном столбце и может использовать ее, а это, в свою очередь, может позволить оптимизировать работу с ними (см. примеры ниже).
- Формат хранения данных изменен с ориентированного на ряды (row-wise) на колоночно-ориентированный (column-wise). Это позволяет добиться более эффективного хранения данных за счет сжатия по отдельным колонкам, а также более эффективной выборки и обработки данных за счет векторных возможностей современных процессоров (SIMD, наборы инструкций SSE 4.1 и SSE 4.2).

DataFrame API и Spark SQL. Датафреймы

- API обработки данных, предоставляемое конечному пользователю, становится SQL-подобным (в том числе возможно использование не только языков программирования Scala, Java, Python, R, но и самого SQL), что имеет ряд преимуществ: оно проще для пользователя и позволяет быстрее ознакомиться с фреймворком; оно увеличивает повторное использование кода за счет функций из стандартной библиотеки Spark SQL и упрощает написание программ обработки; оно служит повышению эффективности программ, так как функции из стандартной библиотеки оптимизированы для работы с новым представлением данных (в отличие от функций, определяемых пользователем, т.е. user-defined function, UDF) и могут, например, использовать встроенную возможность компиляции для конвейера из применяемых последовательно функций.
- При материализации весь полученный сценарий обработки за счет специального встроенного оптимизатора Catalyst приводится из SQL-подобной формы к исполнимой форме графа, состоящей из стадий. В процессе этого приведения применяются различные техники оптимизации, использующие информацию о структуре датафреймов и ее изменении, например: удаление лишних проекций и трансформаций (если результирующие столбцы не используются в конечном датафрейме); сокращение столбцов, читаемых с внешних хранилищ и не использующихся в обработке; осуществление предварительной фильтрации данных на стороне внешнего хранилища, если оно позволяет это (так называемый predicate pushdown), что ведет к уменьшению читаемых с диска данных, уменьшению количества операций сериализации/десериализации, а также уменьшению сетевой передачи данных.

СПАСИБО ЗА ВНИМАНИЕ!