УДК 004.657
doi:10.31799/1684-8853-2022-5-2-11
EDN: AAGGGR

Научные статьи
Articles

Метод ускорения объединения распределенных наборов данных по заданному критерию

Е. С. Тырышкина[a]*, аспирант, orcid.org/0000-0003-4814-4874*
С. Р. Тумковский[a]*, доктор техн. наук, доцент, orcid.org/0000-0002-1647-2171, STumkovskiy@hse.ru*
[a]*Национальный исследовательский университет «Высшая школа экономики», Мясницкая ул., 20, Москва, 101000, РФ*

***Введение:** быстро растущие объемы информации бросают новые вызовы современным технологиям анализа данных. Одной из самых распространенных сопутствующих операций в аналитике является объединение наборов данных. Объединение — крайне ресурсоемкая операция, которая тяжело поддается масштабируемости и повышению эффективности использования ресурсов в распределенных базах данных или системах, основанных на парадигме MapReduce. **Цель:** разработать метод, позволяющий ускорить объединение наборов данных в распределенных системах. **Результаты:** рассмотрены архитектура Apache Spark, особенности распределенных вычислений на основе MapReduce, проанализированы типовые методы объединения наборов данных, рассмотрены основные способы ускорения операции объединения данных. Предложен метод, позволяющий ускорить частный случай объединения, реализованный в Apache Spark с использованием приемов партиционирования и частичной передачи наборов на вычислительные узлы кластера, таким образом, чтобы задействовать одновременно преимущества merge и broadcast объединений. Представленные экспериментальные данные демонстрируют, что метод тем значительнее ускоряет операцию объединения данных относительно стандартных методов, чем больше объем входных данных. Так, для 2 ТБ сжатых данных было получено ускорение до ~37 % в сравнении со стандартным механизмом Spark SQL.*

***Ключевые слова** — распределенные вычисления, MapReduce, Apache Spark, объединение данных.*

Для цитирования: Тырышкина Е. С., Тумковский С. Р. Метод ускорения объединения распределенных наборов данных по заданному критерию. *Информационно-управляющие системы*, 2022, № 5, с. 2–11. doi:10.31799/1684-8853-2022-5-2-11, EDN: AAGGGR
For citation: Tyryshkina Y. S., Tumkovskiy S. R. A method to accelerate the joining of distributed datasets by a given criterion. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2022, no. 5, pp. 2–11 (In Russian). doi:10.31799/1684-8853-2022-5-2-11, EDN: AAGGGR

Введение

В течение многих лет подходы к оптимизации запросов заимствовались из традиционных баз данных и адаптировались к среде распределенных баз данных. Таким образом, знания, полученные при работе с централизованными реляционными системами управления базами данных, были использованы для оптимизации вычислений в распределенных системах. Однако распределенные вычисления обладают уникальными характеристиками, которые часто создают как новые проблемы, так и новые возможности оптимизации.

Одной из важных задач при анализе данных является задача их объединения. В настоящее время она широко изучается, предлагаются новые методы ее решения [1–3], особенно для работы с большими наборами данных в распределенной среде, однако имеющиеся в настоящее время методы недостаточно эффективны: требуют больших перемещений данных между вычислительными нодами кластера, используют много оперативной памяти.

Первая проблема, с которой приходится сталкиваться при выполнении запроса на объединение данных в распределенных базах данных, — это сортировка и перемещение данных внутри кластера, которое увеличивает нагрузку на сеть, что, несмотря на технический прогресс, остается одним из узких мест в кластерных системах. Вторая проблема заключается в том, что невозможно поместить весь набор данных в оперативную память, и данные многократно перезаписываются на жесткие диски, что также сильно замедляет их обработку. Третья проблема вызвана особенностями данных, а именно тем, что некоторые значения в данных встречаются гораздо чаще, чем другие. В этом случае простая стратегия объединения является причиной неравномерного разделения данных на партиции на вычислительных узлах. Затраты на обработку таких данных приводят в лучшем случае к значительному увеличению общего времени решения задачи объединения, а в худшем — к полной невозможности ее решить.

В настоящей работе рассматриваются особенности вычислений в распределенной среде, обсужда-

ется архитектура Apache Spark, для которой предложена стратегия ускорения решения задачи объединения данных. В качестве основного критерия сравнения архитектур Apache Spark и MapReduce использовано время решения задачи объединения данных. При этом надежность вычислений и удобство разработки считаются равными. Показаны возможные проблемы и узкие места типовых стратегий объединения данных и особенности их применения. Представлен разработанный метод решения задачи объединения распределенных наборов данных по заданному критерию и его экспериментальные исследования.

Анализ технологии MapReduce

В настоящее время крупные аналитические платформы ежедневно обрабатывают десятки терабайт данных. Это привело к появлению новых файловых систем и систем управления базами данных [4]. Одной из таких является файловая система Google (GFS) [5] или распределенная файловая система Hadoop (HDFS). Долгое время самым быстрым и надежным решением для создания приложений в этой файловой системе был Google MapReduce, который является основным компонентом Hadoop и обеспечивает обработку данных [6]. Hadoop работает со структурированными, слабоструктурированными и неструктурированными данными.

MapReduce — инструмент распределенных вычислений, способный работать на тысячах узлов кластера [7], является моделью программирования для разработки масштабируемых параллельных приложений с большими данными в вычислительных кластерах [8].

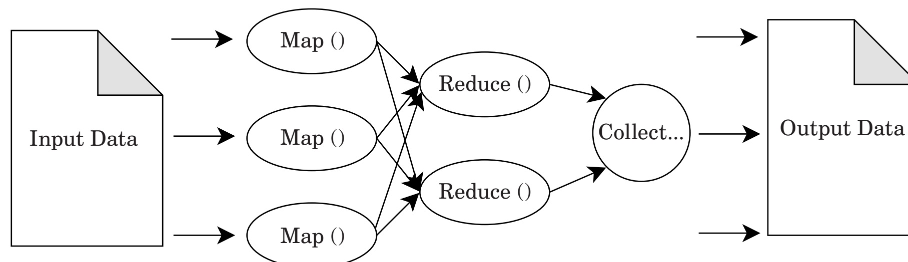
Концепция MapReduce заключается в том, что процесс обработки данных, хранящихся в одном или нескольких файлах, делится на два этапа: отображение и сокращение (рис. 1). Результатом обработки являются пары ключ-значение. Каждый этап имеет ключ-значение в качестве входа и выхода. Ключи и значения не являются внутренними свойствами данных, но они выбираются пользователем, который анализирует данные. Генерация пары ключ-значение Hadoop зависит от набора входных данных и требуемого выхода. Основная задача первого этапа (map) — отфильтровать данные и предоставить возможность группировать данные по ключу, т. е. (k1, v1) → List (k2, v2). Второй этап (reduce) использует функцию для объединения значений, связанных одним ключом, и получения результата для этого ключа, т. е. (k2, List (v2)) → List (k3, v3). Ключ — это поле или объект, по которому будет выполняться группировка и агрегация на этапе reduce. Значение — это поле или объект, к которому будет применяться функция агрегирования.

Данные на выходе из этапа map разделяются и сортируются таким образом, чтобы их можно было распределить в соответствии с критерием группировки на следующей стадии по датанодам, которые будут участвовать в вычислении агрегатной функции на этапе reduce.

Что касается решения задачи объединения данных [9], технология MapReduce гарантирует высокую надежность вычислений. Чрезмерно медленное решение задачи из-за неравномерности ключей в партиционированных наборах данных и многократного повторения операций чтения и записи является недостатком данной технологии [10, 11].

Анализ архитектуры Spark

Apache Spark — это технология кластерных вычислений, получившая развитие на основе технологии MapReduce. В отличие от заложенной в модель MapReduce концепции на основе этапов map и reduce, которая использует дисковое хранилище, в Spark применяется подход поэтапной обработки в оперативной памяти. Это позволяет существенно увеличить скорость обработки в Spark по сравнению с MapReduce [12]. Использование оперативной памяти в процессе вычислений предоставляет возможность многократного доступа к данным, что необходимо для быстрой



■ ***Рис. 1.*** Диаграмма MapReduce
■ ***Fig. 1.*** MapReduce diagram

работы алгоритмов машинного обучения, где широко применяются итерационные алгоритмы. Кеширование промежуточных результатов в памяти ускоряет работу таких алгоритмов.

В основе архитектуры Apache Spark лежат две основные абстракции:

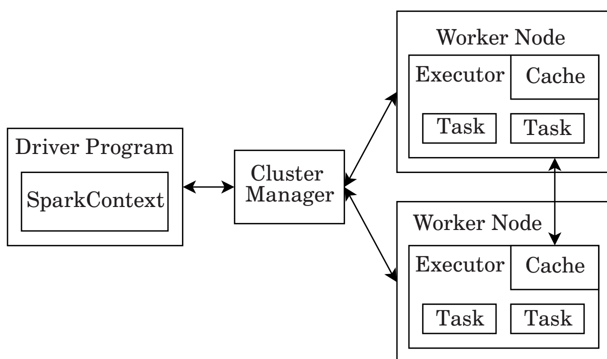
1) направленный ациклический граф (Directed Acyclic Graphs, DAG);

2) устойчивые распределенные наборы данных (Resilient Distributed Datasets, RDD).

Directed Acyclic Graphs — это последовательность вычислений, выполняемых с данными, где каждая вершина является разделом RDD, а ребра — это функция преобразования данных. Здесь под разделами понимаются части данных, лежащих на разных датанодах. В отличие от партиции, данные в разделе RDD не объединяются некоторым общим критерием.

Resilient Distributed Datasets — это интерфейс для работы с коллекцией неизменяемых объектов Java или Scala, которые являются ссылками на разделы набора данных, распределенные по нескольким узлам кластера. В процессе работы разделы могут быть загружены в память на датанодах кластера, а коллекции являются постоянными для того, чтобы их можно было восстановить в случае потери части набора данных. Абстракция DAG помогает устранить многоэтапную модель выполнения Hadoop MapReduce и обеспечивает повышение производительности по сравнению с Hadoop. Весь процесс работы с данными в Spark заключается в применении к данным двух типов операций: преобразования и действия.

Apache Spark использует архитектуру главный/подчиненный с двумя основными программами-демонами (Master Daemon, Slave Daemon) и менеджером кластера (Cluster Manager). В кластере Spark всегда есть один главный (драйвер) и любое количество подчиненных (исполнителей) (рис. 2). Драйвер и исполнители запускают свои отдельные процессы Java, и пользователи могут запускать их в одном кластере Spark или на разных машинах.



■ ***Рис. 2.*** Архитектура Apache Spark [13]

■ ***Fig. 2.*** Apache Spark architecture [13]

Программа-драйвер, исполняемая на главном узле кластера Spark, планирует задание и координирует его с диспетчером кластера. Драйвер преобразует RDD в DAG, разбивает задачу на несколько этапов и сохраняет метаданные обо всех RDD и их разделах.

Исполнитель отвечает за выполнение задач и обрабатывает данные. Исполнитель взаимодействует с системами хранения: читает и записывает данные во внешние источники, сохраняет результаты вычислений в памяти, кеше или на жестких дисках.

При запуске приложения Spark драйвер преобразует пользовательский код в логический DAG. На этом этапе драйвер пытается оптимизировать DAG, на основе которого строится физический план выполнения программы. Программа разбивается на небольшие задачи и отправляется на выполнение в кластер. После этого драйверу необходимо согласовать использование ресурсов с менеджером кластера, который запускает процессы исполнителя на узлах кластера.

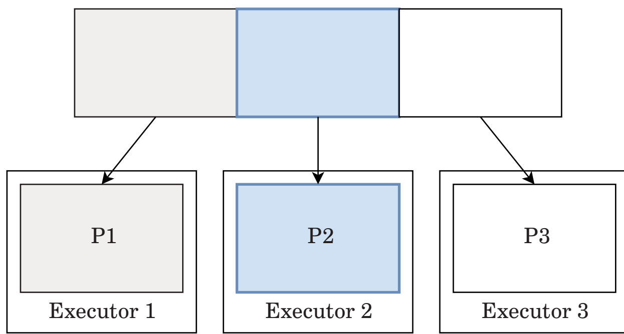
Распространенные методы выполнения операции объединения в Apache Spark

Обычно на операции объединения требуется много ресурсов кластера как сетевых, поскольку они вызывают перераспределение данных внутри кластера, так и вычислительных, чтобы провести сортировку записей и их локальное объединение, т. е. выполнить в памяти много операций сравнения [14]. Если RDD не имеют схемы партиционирования или они различны для каждого набора данных, их необходимо будет создать, что приведет к перемещению данных таким образом, чтобы оба RDD имели одинаковые схемы и данные с одинаковыми ключами находились в одних и тех же партициях (рис. 3).

Как и в случае с большинством операций, ключ-значение, время, затраченное на выполнение операции, нагрузка на сеть между нодами кластера и количество ресурсов памяти увеличиваются с количеством ключей и расстоянием передачи данных, т. е. физической протяженностью маршрута, в том числе количеством маршрутизирующего и регенерирующего оборудования на участке сети, который записи должны преодолеть внутри кластера, чтобы добраться до нужной партиции.

Apache Spark использует в своих вычислениях две стратегии [14]: узел-узел и один узел.

Стратегия объединения узел-узел предполагает, что данные во время операции объедине-



■ *Рис. 3.* Схема идеального партиционирования, при которой данные равномерно разделены на партиции и каждая партиция (P) целиком находится на одном исполнителе (Executor)

■ *Fig. 3.* The scheme of ideal partitioning, in which the data is evenly divided into partitions and each partition (P) is entirely located on one executor

ния будут перемещаться между узлами кластера так, чтобы все ключи оказались на одном узле, в то время как стратегия одного узла выполняет широковещательный тип объединения, когда данные одного набора данных отправляются на каждый узел, т. е. информация дублируется на каждой датаноде в кластере, который уже может сравнивать ключи локально. Эти подходы будут рассмотрены более подробно ниже.

Способы ускорения операции объединения в Apache Spark

В первую очередь необходимо убедиться, что все основные рекомендации по выполнению операции объединения уже выполнены. Перед объединением следует подготовить данные:

— если оба RDD имеют одинаковые ключи партиционирования, то для того, чтобы ускорить операцию, желательно сначала выполнить отдельную операцию агрегирования (combByKey, cogroup);

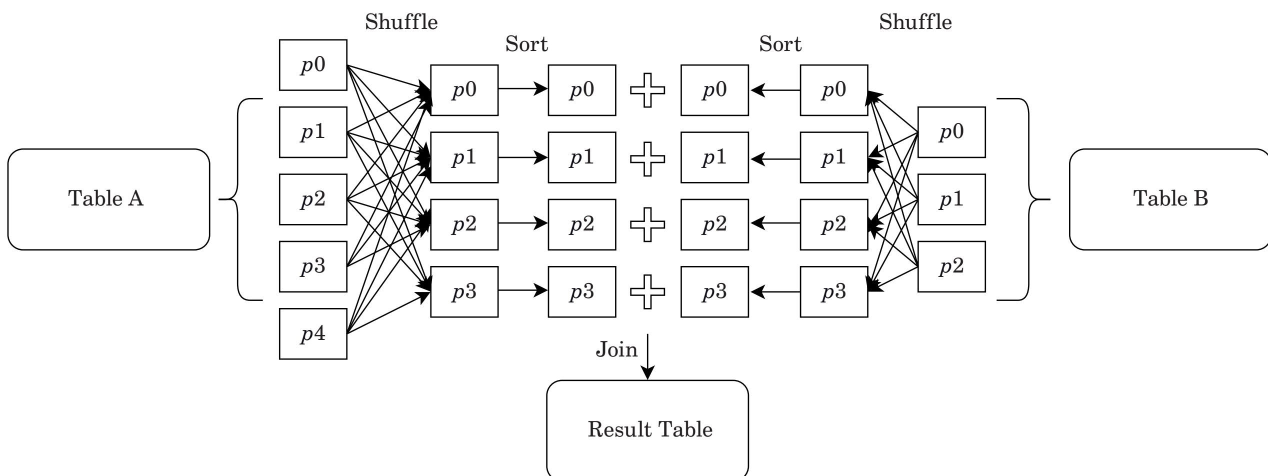
— если один RDD имеет некое легко идентифицируемое подмножество ключей, которые вы в конечном итоге не будете использовать, то лучше их отфильтровать перед объединением;

— предварительно репартиционировать данные, если это возможно.

Выделяются три основные стратегии объединения: Sort-Merge, Broadcast и Shuffle Hash.

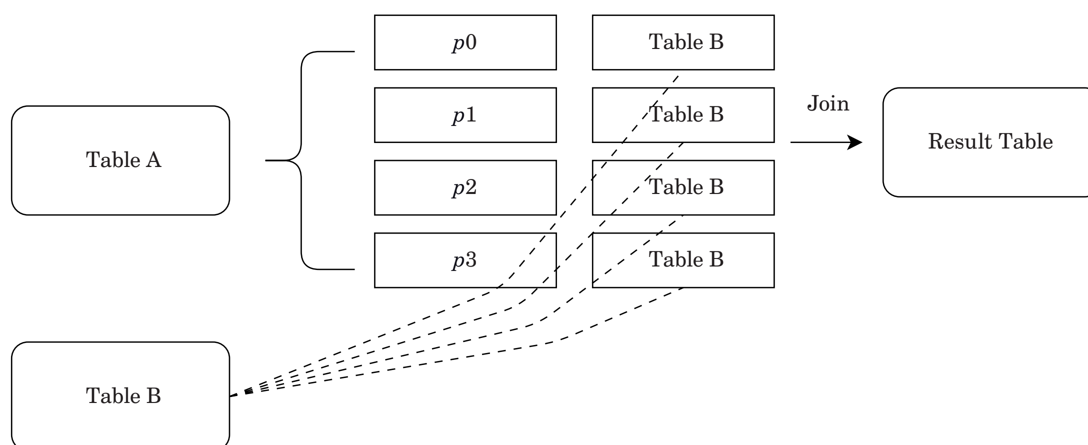
Объединение Sort-Merge (рис. 4) состоит из двух этапов и является предпочтительным, поскольку может записывать данные на диски, не используя хранение в оперативной памяти. На первом этапе сортируются наборы данных, что само по себе занимает много времени. Ускорение данного этапа исследовалось в работах [15–17]. На втором этапе отсортированные данные объединяются в партиции и поэлементно сравниваются по ключу. В Spark выше версии 2.3 Sort-Merge используется в качестве алгоритма по умолчанию. Однако его использование по умолчанию можно отключить с помощью параметра конфигурации spark.sql.join.preferSortMergeJoin.

Объединение Broadcast (рис. 5) обеспечивает максимальную производительность среди стандартных методов и решает проблемы неравномерного сегментирования и ограниченного параллелизма, однако он актуален только для небольших наборов данных. Чтобы применить этот подход, размер таблицы должен быть меньше значения, настроенного с помощью конфигурации контекста spark.sql.autoBroadcastJoinThreshold (по умолчанию 10 МБ).



■ *Рис. 4.* Объединение Sort-Merge

■ *Fig. 4.* Sort-Merge join diagram



■ ***Рис. 5.*** Объединение Broadcast
■ ***Fig. 5.*** Broadcast join diagram

Объединение Shuffle Hash (рис. 6) работает на основе концепции MapReduce. На этапе map на основе данных вычисляются ключи, по которым будет выполняться объединение. Эти значения используются как ключи партиции, а данные перемещаются внутри кластера так, чтобы записи, относящиеся к одному ключу объединения, были расположены на одной датаноде. Spark выбирает Shuffle Hash объединение, когда невозможно применить Sort-Merge, что проверяется функцией:

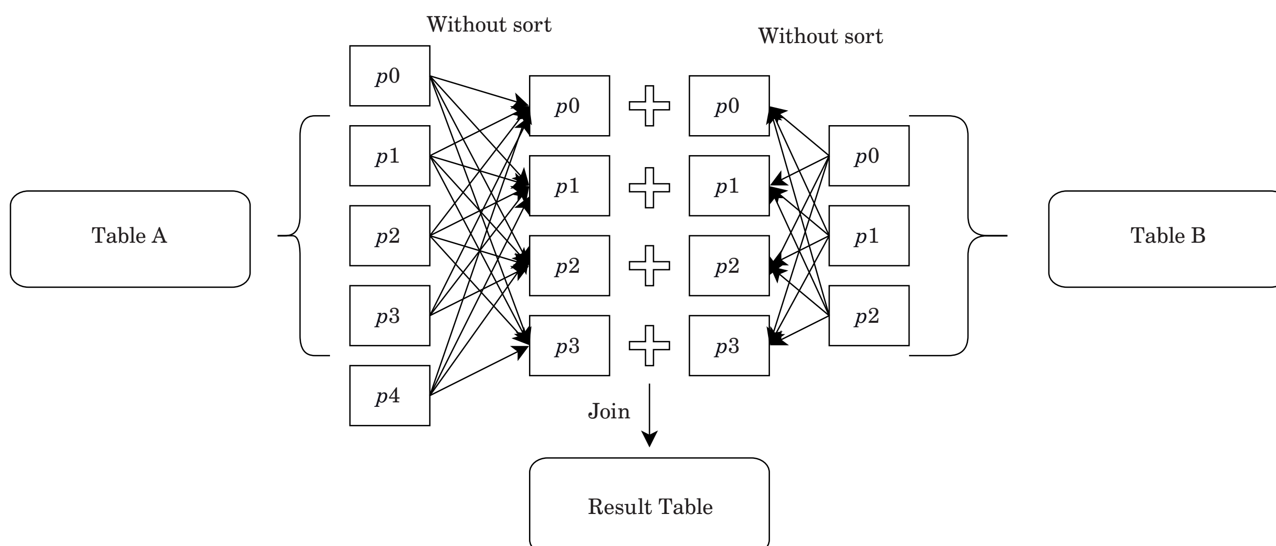
```
def canBuildLocalHashMap (plan: LogicalPlan): Boolean = {
    plan.statistics.sizeInBytes < conf.autoBroadcastJoinThreshold
* conf.numShufflePartitions
}
```

Создание хеш-таблиц — сложная операция, и ее можно выполнить, только если средний размер одной партиции достаточно мал, настолько, чтобы поместиться в оперативную память вычислительной машины, на которой производится расчет.

На практике в обычном режиме использования кластера производительность также зависит от конфигурации Spark, посторонней нагрузки на кластер в момент расчета и синергии между конфигурацией и кодом.

Метод ускорения операции объединения

Предлагаемый в работе метод сочетает в себе преимущества вышеперечисленных подходов. В то же время он не требует, чтобы наборы данных полностью помещались в ОЗУ и проводи-



■ ***Рис. 6.*** Объединение Shuffle Hash
■ ***Fig. 6.*** Shuffle Hash diagram

лась их сортировка, что обеспечивает простоту и более высокую скорость выполнения операции объединения. Допустим, у нас есть два несортированных набора данных — RDD1 и RDD2. Необходимо позаботиться о том, чтобы набор RDD1 хранился на диске таким образом, чтобы данные каждой партиции записывались в отдельный файл. Это можно сделать, описав свой собственный класс customPartitioner, который задает некоторую логику партиционирования, например, вычисляя хеш-функцию по критерию объединения. Однако при слишком большом количестве партиций планирование задач займет больше времени, чем фактический расчет, а при слишком маленьком будет низкая параллельность расчета. Важно равномерно распределить набор данных по партициям и предотвратить неравномерное их разделение, когда большой объем данных концентрируется на одной или нескольких машинах в кластере [18]. Допустимый для работы метода процент неравномерности данных ограничивается физическими ресурсами кластера. Данных в любой партиции всегда должно быть меньше, чем имеется оперативной памяти на узлах кластера. Неравномерное попадание записей в одну партицию может привести к ошибкам OOM (Out of Memory) на исполнителях. Также если данные сильно неравномерны по ключу [19], то один исполнитель может работать дольше, чем остальные, и задержать общее выполнение задачи. Таким образом, неравномерность распределения данных приводит к пропорциональному увеличению времени расчета. В данном случае проблема неравномерности решалась путем прогнозирования и распределения проблемных ключей на основе информации, предварительно полученной из данных, и наложения на них дополнительной функции партиционирования. Причина неравномерности данных всегда обусловлена семантикой, источником данных и критерием объединения. Поэтому в процессе решения задачи объединения стоит исследовать, насколько равномерно данные группируются по критерию будущего объединения. Пример того, как в Spark можно записать партиции в один файл:

```
RDD.map{ line => someFunc
}.partitionBy(customPartitioner).saveAsTextFile(filePath)
```

Чтобы исполнители самостоятельно скачивали файл из HDFS, необходимо передать конфигурацию Hadoop на каждый узел, участвующий в расчетах:

```
val confBroadcast = sc.broadcast(new SerializableWritable(sc.hadoopConfiguration))
def readFromHDFS(configuration: Configuration, path: String): String = {
    val fs: FileSystem = FileSystem.get(configuration)
    val inputStream = fs.open(new Path(path));
    val writer = new StringWriter();
    IOUtils.copy(inputStream, writer, "UTF-8");
    writer.toString
}
```

Ключевым преимуществом этого подхода является отсутствие этапа shuffle, задача которого — промежуточное перемещение данных внутри кластера с целью сконцентрировать данные на основе ключа партиции на одном узле, т. е. физическое перемещение данных между нодами кластера во время разделения данных на партиции. Операция `saveAsTextFile` при RDD1 производится без этапа shuffle, а партиционирование RDD2 существует только на логическом уровне программы.

Таким образом:

— RDD1 делится на партиции путем записи данных каждой партиции в отдельный файл;

— RDD2 никак не перемещается внутри кластера, так как объекты этого RDD ссылаются на партиции локально на каждой датаноде. Строки, принадлежащие одной партиции данного датасета, не концентрируются на одном узле, а остаются распределенными по разным узлам.

Поскольку известно, в каком файле находятся строки определенной партиции RDD1, то мы можем прочитать этот файл напрямую на том узле, где находится соответствующая партиция RDD2. Преимущество метода в скорости получается за счет того, что, в отличие от стандартных стратегий объединения, в нашем случае исключается промежуточный этап перемещения данных между узлами, а между нодами кластера перемещается только часть данных на объединение — RDD1, в то время как RDD2 физически не перемещается.

Для работы с партициями используются функции mapPartitions() или mapPartitionsWithIndex(), которые преобразуют каждую партицию исходного набора данных в несколько элементов результата (в некоторых случаях ни в один). Одним из важных вариантов использования этих функций может быть некоторая тяжелая инициализация, которая делается один раз для нескольких элементов и повторно используется для каждого потока/партиции, в отличие от функции map(), применяемой к каждому элементу в наборе данных.

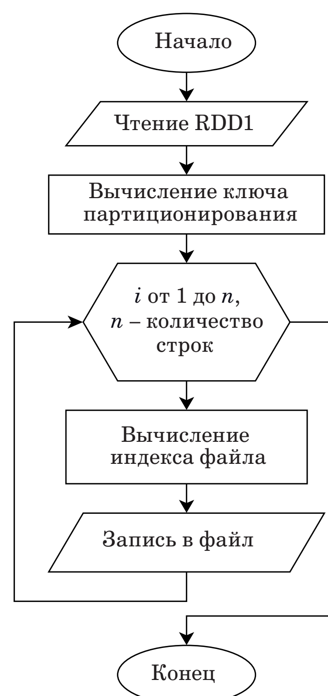
Затем в процессе обработки RDD2 можно произвести переразбиение по тому же алгоритму, что и в RDD1 (с помощью customPartitioner), а затем внутри функции mapPartitionsWithIndex() обратиться по индексу к конкретному файлу первого набора и загрузить его в память напрямую с использованием структуры данных «словарь». Словари реализованы в виде хеш-таблиц, где операции поиска выполняются за постоянное

время, так как элемент запрашивается напрямую, без проведения сравнений, т. е. сложность алгоритма поиска соответствует O(1):

```
val data = RDD2.partitionBy(customPartitioner) val res = data.
mapPartitionsWithIndex((index, partition) = {
val conf = confBroadcast.value.value
val partitionRDD1 = readFromHDFS(conf, path + index)
val partitionData = partitionRDD1.map {
case Array(k, v) => k -> v.toLong;
case _ => “” -> “”}
}.toMap
val newPartition = partition.map(
record => {(record , partitionData(record._1))})
new Partition
})
```

Важно помнить, что процедура переразбиения выполнятся вне контекста Spark. Spark не сможет отслеживать и предоставлять статистику загрузки данных, поэтому разработчик должен самостоятельно контролировать использование оперативной памяти и памяти сборщика мусора на исполнителях. Без хорошего понимания объема данных и доступных вычислительных ресурсов этот подход будет невозможно применить.

Основная особенность этого подхода заключается в том, что каждый исполнитель сможет самостоятельно загружать данные в ОЗУ датаноды, на которой он находится, с любой другой машины в кластере, без участия машины-драйвера, что является своего рода «узким местом». С другой стороны, будет загружен блок данных, который действительно нужен в данный момент конкретному исполнителю для тех партиций, на которых он выполняет вычисления, что позволяет своевременно загружать в память часть набора данных, а не целиком весь набор. В некотором смысле это имитация поведения «объединение слиянием», но в нашем случае нет необходимости сортировать строки, что также дает серьезный выигрыш в скорости выполнения.



■ ***Рис. 7.*** Алгоритм первого этапа
■ ***Fig. 7.*** Algorithm of the first stage

Алгоритм предлагаемого метода и результаты экспериментов

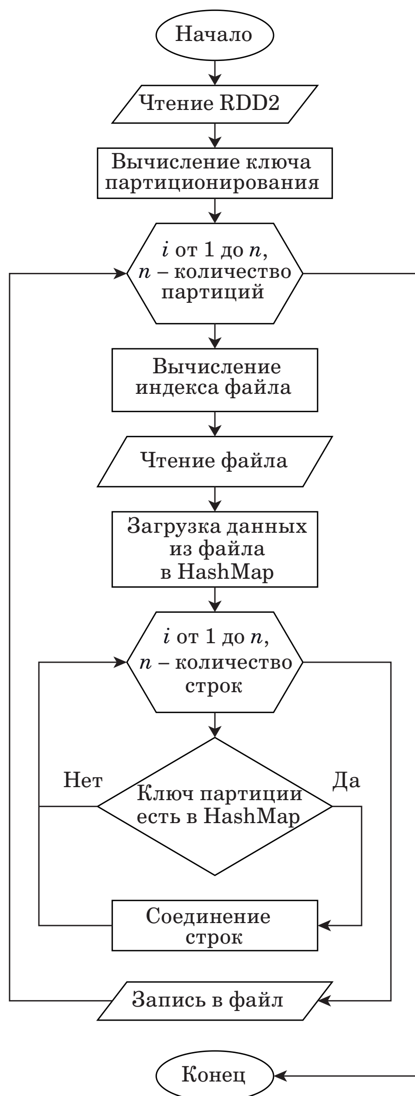
Алгоритмически предложенный метод реализуется в два этапа. На первом этапе (рис. 7) реорганизуется хранение первого набора данных (RDD1) параллельным чтением строк из одних файлов и записью в другие. Синхронизация доступа к файлам обеспечивается файловой системой HDFS. Оригинальные данные не изменяются, только сортируются. Попадание определенной записи в файл выбирается на основе критерия объединения. Данная операция выполняется без промежуточного этапа перемещения данных между стадиями map и reduce.

На втором этапе (рис. 8) выполняется репартиционирование на логическом уровне также на основе заданного критерия объединения без физического перемещения данных по кластеру. Далее для каждой партиции данных параллельно выполняются чтение и загрузка в память файлов первого набора RDD1, где уже локально производится операция сравнения ключей строк, последующее соединение и запись результата.

Разработка во фреймворке Spark доступна на языках Python и R, имеющих большую популярность среди аналитиков. На языках Java и Scala написано ядро фреймворка Spark, поэтому они являются «родными» для него. В отличие от других, язык программирования Scala для Spark предоставляет доступ к новейшим и наиболее важным функциям. Поэтому для реализации предложенного алгоритма выбран язык Scala.

Экспериментальные исследования разработанного метода и его алгоритмической реализации проводились с целью определить время решения задачи объединения неструктурированных текстовых данных объемом от 100 ГБ до 7 ТБ.

В эксперименте были задействованы текстовые данные, сжатые с помощью алгоритма GZIP, 128 программ-исполнителей с шестью ядрами и 16 ГБ памяти и программа драйвера с 8 ГБ памяти. Исходные данные делились равномерно по ключу партиционирования на 3500 партиций. Время объединения данных измерялось стандартной программой spark.time(). Каждое измерение проводилось пять раз, а время выполнения усреднялось.



■ ***Рис. 8.*** Алгоритм второго этапа
■ ***Fig. 8.*** Algorithm of the second stage

Результаты исследования предложенного в работе метода объединения в сравнении со стандартным Sort-Merge объединением, используемым в Spark SQL [20], представлены на рис. 9.

По результатам проведенных исследований можно сделать вывод, что предложенный метод и его алгоритмическая реализация позволяют решать задачу объединения неструктурированных текстовых данных объемом 2 ТБ на 37 % быстрее, а для данных объемом 7 ТБ примерно на 47 %.

Заключение

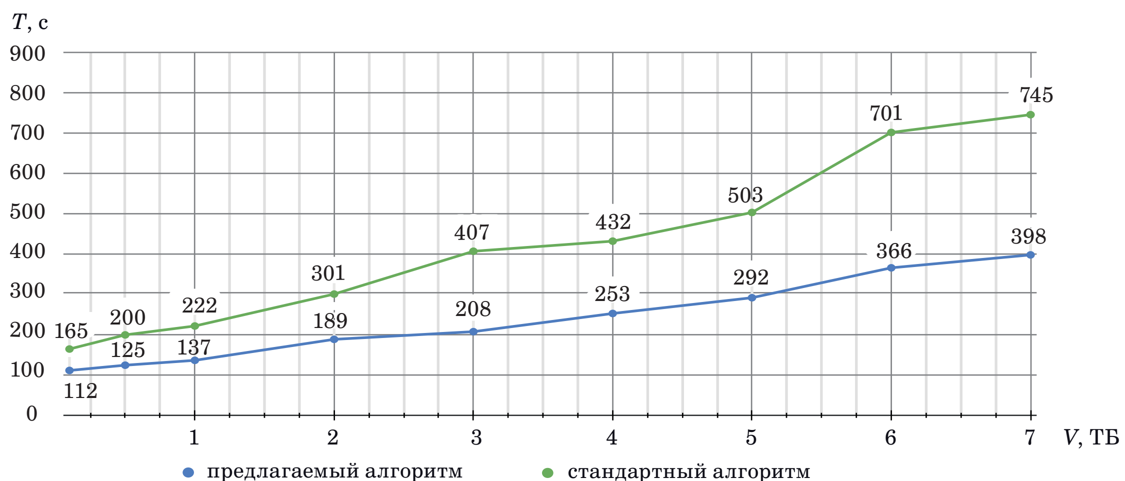
Разработка быстрых и надежных методов обработки больших данных в распределенных системах является актуальной задачей, а имеющиеся методы недостаточно эффективны. Среди них технология Spark является наиболее эффективной с точки зрения скорости вычислений.

Одним из узких мест в кластерных системах являются задачи сортировки и перемещения данных внутри кластера, увеличивающие нагрузку на сеть.

Другой важной задачей при анализе данных является их объединение, методы которого, включая технологию Spark, на сегодняшний момент по критерию времени выполнения недостаточно эффективны.

Предлагаемый метод и его алгоритмическая разработка в сравнении с технологией Spark эффективнее при решении задачи объединения данных на 37–47 %.

Дальнейшее развитие предложенного метода предполагает его модификацию для объединений разных типов данных, построение оптимальных алгоритмов распределения наборов данных по партициям с целью минимизировать их неравномерное разделение.



■ ***Рис. 9.*** Время выполнения расчета предложенным и стандартным методами
■ ***Fig. 9.*** Time to complete the calculation by the proposed and standard methods

Литература

1. **Yoshimi M., Oge Y., Yoshinaga T.** Pipelined parallel join and its FPGA-based acceleration. *Proc. of the ACM Transactions on Reconfigurable Technology and Systems*, 2007, vol. 10, iss. 4, no. 28, pp 1–28. doi:10.1145/3079759
2. **Zhang X., Chen L., Wang M.** Efficient multi-way theta-join processing using MapReduce. *Proc. of the VLDB Endowment*, 2012, vol. 5, no. 11, pp. 1184–1195. doi:10.14778/2350229.2350238
3. **Saleem M., Potocki A., Soru T., Hartig O., Axel-Cyrille Ngonga Ngomo.** CostFed: Cost-based query optimization for SPARQL endpoint federation. *Proc. of the 14th Intern. Conf. on Semantic Systems*, 2018, Austria, pp. 163–174. doi:10.1016/j.procs.2018.09.016
4. **Sudhaka M., Satheesh N., Balu S., Reddy A. M., Murugan G.** Optimizing joins in a Map-Reduce for data storage and retrieval performance analysis of query processing in HDFS for big data. *Intern. Journal of Advanced Trends in Computer Science and Engineering*, 2019, pp. 2062–2087. doi:10.30534/ijatcse/2019/33852019
5. **Elkawkagy M., Elbeh H.** High performance hadoop distributed file system. *Intern. Journal of Networked and Distributed Computing*, 2020, vol. 8, iss. 3, pp. 119–123. doi:10.2991/ijndc.k.200515.007
6. **Maitrey S., Jha C. K.** MapReduce: Simplified data analysis of big data. *Procedia Computer Science*, 2015, vol. 57, pp. 563–571. doi:10.1016/j.procs.2015.07.392
7. **Malleswari T. Y. J. N., Vadivu G.** MapReduce: A technical review. *Indian Journal of Science and Technology*, 2016, vol. 9, iss. 1, pp. 1–6. doi:10.17485/ijst/2016/v9i1/78964
8. **Shim K.** MapReduce algorithms for big data analysis. *Proc. of the VLDB Endowment*, 2012, vol. 5, no. 12, pp. 2016–2017. doi:10.14778/2367502.2367563
9. **Al-Badarneh A. F., Rababa S. A.** An analysis of two-way equi-join algorithms under MapReduce. *Journal of King Saud University "Computer and Information Sciences"*, 2022, vol. 34, iss. 4, pp. 1074–1085. doi:10.3233/JIFS-201220
10. **Gavagsaz E., Rezaee A., Javadi H. H. S.** Load balancing in join algorithms for skewed data in MapReduce systems. *The Journal of Supercomputing*, 2019, vol. 75, pp. 228–254. doi:10.1145/3335484.3335495
11. **Kwon Y. C., Balazinska M., Howe B., Rolia J.** A study of skew in mapreduce applications. *Proc. of the 5th Open Cirrus Summit*, 2011, Moscow.
12. **Gousios G.** Big data software analytics with Apache Spark. *Proc. of the 40th Intern. Conf. on Software Engineering*, 2018, pp. 542–543. doi:10.1145/3183440.3183458
13. **Karau H., Warren R.** *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly, 2017. 342 p.
14. **Bruno N., Kwon Y., Wu M.** Advanced join strategies for large-scale distributed computation. *Proc. of the VLDB Endowment*, 2014, vol. 7, no. 13, pp. 1484–1495. doi:10.14778/2733004.2733020
15. **Saitoh M., Elsayed E. A., Van Chu T., Mashimo S., Kise K.** A high-performance and cost-effective hardware merge sorter without feedback datapath. *IEEE 26th Annual Intern. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 197–204. doi:10.1587/transfun.2021EAL2103
16. **Papaphilippou P., Brooks C., Luk W.** FLiMS: Fast lightweight merge sorter. *Proc. of the 29th Intern. Conf. on Field-Programmable Technology (FPT)*, 2018, pp. 78–85. doi:10.48550/arXiv.2112.05607
17. **Papaphilippou P., Pirk H., Luk W.** Accelerating the merge phase of sort-merge join. *Proc. of the 29th Intern. Conf. on Field-Programmable Logic and Applications (FPL)*, 2019, pp. 100–105. doi:10.1109/FPL.2019.00025
18. **Bandle M., Giceva J., Neumann T.** To partition, or not to partition, that is the join question in a real system. *Proc. of the 2021 Intern. Conf. on Management of Data*, 2021, pp. 168–180. doi:10.1145/3448016.3452831
19. **Memarzia P., Ray S., Bhavsar V.** On improving data skew resilience in main-memory hash joins. *Proc. of the 22nd Intern. Database Engineering & Applications Symp.*, 2018, pp. 226–235. doi:10.1145/3216122.3216156
20. **Armbrust M., Xin R. S., Lian C., Huai Y., Liu D., Bradley J. K., Meng X., Kaftan T., Franklin M. J., Ghodsi A., Zaharia M.** Spark SQL: Relational data processing in spark. *Proc. of the Intern. Conf. on Management of Data (SIGMOD'15)*, 2015, Australia, pp. 1383–1394. doi:10.1145/2723372.2742797

UDC 004.657
doi:10.31799/1684-8853-2022-5-2-11
EDN: AAGGGR

A method to accelerate the joining of distributed datasets by a given criterion

Y. S. Tyryshkina[a], Post-Graduate Student, orcid.org/0000-0003-4814-4874
S. R. Tumkovskiy[a], Dr. Sc., Tech., Associate Professor, orcid.org/0000-0002-1647-2171, STumkovskiy@hse.ru
[a]National Research University «Higher School of Economics», 20, Myasnitskaya St., 101000, Moscow, Russian Federation

Introduction: Rapidly growing volumes of information pose new challenges to modern data analysis technologies. One of the most common related operations in analytics is the joins of datasets. A join is a highly resource-intensive operation which is difficult to scale and improve in terms of resource efficiency in distributed databases or systems based on the MapReduce paradigm. **Purpose:** To develop a method to accelerate the integration of datasets in distributed systems. **Results:** The architecture of Apache Spark, features of distributed computing based on MapReduce are examined, typical methods of joining datasets are analyzed, and the main ways to speed up the operation of joining data are considered. We propose a method to speed up a special case of joins implemented in Apache Spark with the use of partitioning and partial transfer of datasets to cluster computing nodes. The method is realized to simultaneously harness the benefits of merge and broadcast joins. The presented experimental data demonstrate that the larger the volume of input data the more efficiently the method speeds up the operation of joining data in comparison with standard methods. Thus, for 2 TB compressed data, acceleration up to ~37% was obtained in comparison with standard Spark SQL.
Keywords — distributed computing, MapReduce, Apache Spark, joining data.

For citation: Tyryshkina Y. S., Tumkovskiy S. R. A method to accelerate the joining of distributed datasets by a given criterion. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2022, no. 5, pp. 2–11 (In Russian). doi:10.31799/1684-8853-2022-5-2-11, EDN: AAGGGR

References

1. Yoshimi M., Oge Y., Yoshinaga T. Pipelined parallel join and its FPGA-based acceleration. *Proc. of the ACM Transactions on Reconfigurable Technology and Systems*, 2007, vol. 10, iss. 4, no. 28, pp 1–28. doi:10.1145/3079759
2. Zhang X., Chen L., Wang M. Efficient multi-way theta-join processing using MapReduce. *Proc. of the VLDB Endowment*, 2012, vol. 5, no. 11, pp. 1184–1195. doi:10.14778/2350229.2350238
3. Saleem M., Potocki A., Soru T., Hartig O., Axel-Cyrille Ngonga Ngomo. CostFed: Cost-based query optimization for SPARQL endpoint federation. *Proc. of the 14th Intern. Conf. on Semantic Systems*, 2018, Austria, pp. 163–174. doi:10.1016/j.procs.2018.09.016
4. Sudhaka M., Satheesh N., Balu S., Reddy A. M., Murugan G. Optimizing joins in a Map-Reduce for data storage and retrieval performance analysis of query processing in HDFS for big data. *Intern. Journal of Advanced Trends in Computer Science and Engineering*, 2019, pp. 2062–2087. doi:10.30534/ijatcse/2019/33852019
5. Elkawkagy M., Elbeh H. High performance hadoop distributed file system. *Intern. Journal of Networked and Distributed Computing*, 2020, vol. 8, iss. 3, pp. 119–123. doi:10.2991/ijndc.k.200515.007
6. Maitrey S., Jha C. K. MapReduce: Simplified data analysis of big data. *Procedia Computer Science*, 2015, vol. 57, pp. 563–571. doi:10.1016/j.procs.2015.07.392
7. Malleswari T. Y. J. N., Vadivu G. MapReduce: A technical review. *Indian Journal of Science and Technology*, 2016, vol. 9, iss. 1, pp. 1–6. doi:10.17485/ijst/2016/v9i1/78964
8. Shim K. MapReduce algorithms for big data analysis. *Proc. of the VLDB Endowment*, 2012, vol. 5, no. 12, pp. 2016–2017. doi:10.14778/2367502.2367563
9. Al-Badarneh A. F., Rababa S. A. An analysis of two-way equi-join algorithms under MapReduce. *Journal of King Saud University "Computer and Information Sciences"*, 2022, vol. 34, iss. 4, pp. 1074–1085. doi:10.3233/JIFS-201220
10. Gavagsaz E., Rezaee A., Javadi H. H. S. Load balancing in join algorithms for skewed data in MapReduce systems. *The Journal of Supercomputing*, 2019, vol. 75, pp. 228–254. doi:10.1145/3335484.3335495
11. Kwon Y. C., Balazinska M., Howe B., Rolia J. A study of skew in mapreduce applications. *Proc. of the 5th Open Cirrus Summit*, 2011, Moscow.
12. Gousios G. Big data software analytics with Apache Spark. *Proc. of the 40th Intern. Conf. on Software Engineering*, 2018, pp. 542–543. doi:10.1145/3183440.3183458
13. Karau H., Warren R. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly, 2017. 342 p.
14. Bruno N., Kwon Y., Wu M. Advanced join strategies for large-scale distributed computation. *Proc. of the VLDB Endowment*, 2014, vol. 7, no. 13, pp. 1484–1495. doi:10.14778/2733004.2733020
15. Saitoh M., Elsayed E. A., Van Chu T., Mashimo S., Kise K. A high-performance and cost-effective hardware merge sorter without feedback datapath. *IEEE 26th Annual Intern. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 197–204. doi:10.1587/transfun.2021EAL2103
16. Papaphilippou P., Brooks C., Luk W. FLiMS: Fast lightweight merge sorter. *Proc. of the 29th Intern. Conf. on Field-Programmable Technology (FPT)*, 2018, pp. 78–85. doi:10.48550/arXiv.2112.05607
17. Papaphilippou P., Pirk H., Luk W. Accelerating the merge phase of sort-merge join. *Proc. of the 29th Intern. Conf. on Field-Programmable Logic and Applications (FPL)*, 2019, pp. 100–105. doi:10.1109/FPL.2019.00025
18. Bandle M., Giceva J., Neumann T. To partition, or not to partition, that is the join question in a real system. *Proc. of the 2021 Intern. Conf. on Management of Data*, 2021, pp. 168–180. doi:10.1145/3448016.3452831
19. Memarzia P., Ray S., Bhavsar V. On improving data skew resilience in main-memory hash joins. *Proc. of the 22nd Intern. Database Engineering & Applications Symp.*, 2018, pp. 226–235. doi:10.1145/3216122.3216156
20. Armbrust M., Xin R. S., Lian C., Huai Y., Liu D., Bradley J. K., Meng X., Kaftan T., Franklin M. J., Ghodsi A., Zaharia M. Spark SQL: Relational data processing in spark. *Proc. of the Intern. Conf. on Management of Data (SIGMOD'15)*, 2015, Australia, pp. 1383–1394. doi:10.1145/2723372.2742797