МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

ОТКАЗОУСТОЙЧИВОСТЬ В СИСТЕМАХ ПОТОКОВОЙ ОБРАБОТКИ ДАННЫХ

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

Студентки 5 курса 551 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Зотовой Тамары Давидовны

Научный руководитель
к. ф.-м. н., доцент _____ А. А. Кузнецов

Заведующий кафедрой
к. ф.-м. н., доцент _____ А. С. Иванов

Саратов 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Системы потоковой обработки данных, как и другие информационные система, могут быть подвержены сбоям. Очевидно, что системы должны обладать механизмом отказоустойчивости и быть способными оперативно восстанавливаться после любого возможного отказа, нивелируя негативные последствия для бизнеса. Процессы таких систем в большей своей части хранят некоторое состояние вычислений (например, оконные агрегирования такие как счетчики, вычисления средних значений, гистограммы) [1], так что при восстановлении системы должно гарантироваться восстановление согласованного состояния всей системы до сбоя, чтобы обеспечить корректное продолжение потоковой обработки данных.

Целью данной работы является обзорное рассмотрение различных подходов к реализации отказоустойчивости в системах потоковой обработки больших данных с акцентом на механизмы восстановления путем отката системы до ранее достигнутого согласованного состояния всех процессов системы. Для достижения данной цели были поставлены следующие задачи:

- ознакомление с теоретическими аспектами построения систем потоковой обработки данных с акцентом на их ключевые функции, оказывающие влияние на восстановление после всевозможных отказов;
- обзорное рассмотрение различных подходов к восстановлению систем путем отката с особым вниманием к подходам, основанным на контрольных точках;
- рассмотрение механизма отказоустойчивости, реализованного в фреймворке Apache Flink;
- решение проблемы с сохранением несогласованного состояния системы в контрольной точке для итерационных заданий в Apache Flink. [2]

1 Потоковая обработка данных

Термин «потоковая обработка данных» (stream processing) подразумевает обработку любых данных в виде потока, однако сейчас под ним принято понимать скорее обработку некоторых пакетов данных, таких как сообщения, но не обработку аудио и видео данных. В данной работе идет речь об области обработки больших данных как потоков.

1.1 Ключевые функции систем потоковой обработки

1.1.1 Семантика доставки сообщений

В общем случае принято выделять следующие семантики доставки сообщений в звене анализа:

- Не менее одного раза, т.е. сообщение не может потеряться, но может быть обработано несколько раз;
- Ровно один раз, эта семантика означает, что сообщение не может потеряться и обрабатывается ровно один раз;
- Не более одного раза — сообщение может потеряться, но никогда не может быть обработано дважды.

Необходимый уровень семантики доставки сообщений зависит от конкретной решаемой задачи. При ее выборе необходимо учитывать критичность потери или дублирования сообщений при обработке данных.

1.1.2 Управление состоянием

Если потоковый алгоритм анализа становится сложнее, чем операции с одним лишь текущим сообщением без учета зависимостей от прошлых сообщений и внешних данных, то возникает необходимость в запоминании состояния и, вполне вероятно, понадобится служба управления состоянием, предоставляемая выбранной системой.

Системы можно разделить в зависимости от предлагаемых средств управления состоянием:

- системы с хранением состояния в памяти,
- системы, предлагающие постоянное распределенное хранилище.

Первые системы применимы только в том случае, если сбой системы и, как следствие, потеря текущего состояния не являются критичными. Такие системы могут использоваться для запоминания состояния вычислений, например

текущего счетчика. Вторые системы позволяют отвечать на более сложные вопросы и решать широкий круг задач, например сливать два потока данных.

1.1.3 Отказоустойчивость

Способность системы потоковой обработки продолжать функционирование в условиях сбоев — прямое следствие реализованных в ней механизмов отказоустойчивости.

В системах потоковой обработки все распространенные методы обработки ошибок сводятся к той или иной форме дублирования и координации. Обычно потоковый диспетчер дублирует состояние вычисления (потоковой задачи) на различных потоковых процессорах. В случае отказа потоковый диспетчер должен координировать копии, чтобы правильно восстановиться после ошибки. Механизмы отказоустойчивости проектируются в расчете на максимальное число одновременных сбоев.

В общем случае существует два подхода к дублированию и координации в потоковой системе: конечный автомат и восстановление путем отката. Оба подхода предполагают, что алгоритм обработки потока однозначно воспроизводим, т.е. если два правильно работающих потоковых процессора получают одинаковые входные данные в одном и том же порядке, то они порождают одинаковые результаты в одном и том же порядке. В рамках данной работы наибольший интерес представляет именно восстановление путем отката.

2 Восстановление потоковой системы путем отката

Восстановление путем отката рассматривает распределенную систему как совокупность процессов приложения, которые взаимодействуют через сеть. Процессы имеют доступ к надежному устройству хранения, которое устойчиво ко всевозможным сбоям. Процессы обеспечивают отказоустойчивость благодаря использованию этого устройства для периодического сохранения необходимой для восстановления информации во время безотказного выполнения. В случае отказа вышедший из строя процесс использует сохраненную информацию для перезапуска вычислений из промежуточного состояния, тем самым уменьшая количество потерянных вычислений. Информация, используемая для восстановления, включает в себя как минимум состояния участвующих процессов, которые называются контрольными точками. Некоторые протоколы восстановления могут требовать такую дополнительную информацию, как журналы взаимодействий с устройствами ввода и вывода, события, которые происходят с каждым процессом, и сообщения, которыми обмениваются процессы.

Потоковые системы имеют отдельные сложности, связанные с восстановлением путем отката, поскольку обрабатываемые сообщения вызывают межпроцессные зависимости во время безотказной работы. В случае сбоя одного или нескольких процессов в системе эти зависимости могут привести к принудительному откату некоторых процессов, что создает так называемое *распространение отката*. Чтобы понять, почему происходит распространение отката, рассмотрим ситуацию, когда отправитель сообщения m откатывается до состояния, которое предшествует отправке m. Получатель m также должен откатиться до состояния, предшествующего получению m. В противном случае состояния двух процессов были бы несогласованными, поскольку они показывали бы, что сообщение m было получено без отправки, что невозможно при корректном выполнении программы. В некоторых сценариях распространение отката может распространиться до исходного состояния вычислений, при этом теряя всю выполненную до сбоя работу. Такая ситуация известна как *эффект домино*.

Эффект домино может возникнуть, если каждый процесс создает свои контрольные точки независимо от других процессов. Очевидно, что эффект домино является нежелательным, и поэтому для его предотвращения было

разработано несколько методов. Одним из таких методов является применение скоординированных между процессами контрольных точек, чтобы сохранить согласованное состояние всей системы. Этот согласованный набор контрольных точек может затем использоваться для ограничения распространения отката. Альтернативный подход использует контрольные точки, вызванные коммуникацией. При таком подходе каждый процесс создает контрольные точки на основе информации, связанной с сообщениями приложения, полученными от других процессов. Контрольные точки создаются таким образом, что в надежном хранилище всегда существует согласованное для всей системы состояние, что позволяет избежать эффекта домино.

Вышеприведенные подходы реализуют восстановление путем отката на основе контрольных точек, которое опирается только на контрольные точки для обеспечения отказоустойчивости системы. Существует также другой подход, а именно восстановление путем отката на основе журнала, которое объединяет контрольные точки с протоколированием событий. Данный подход особенно привлекателен для приложений, которые часто взаимодействуют с внешним миром, состоящим из устройств ввода и вывода, которые не могут выполнить откат. [3]

3 Фреймворк Apache Flink и механизм отказоустойчивости в нем

Apache Flink является платформой с открытым исходным кодом для распределенной обработки потоковых и пакетных данных. Программы Apache Flink после запуска сопоставляются потокам данных. Каждый поток данных Apache Flink начинается с одного или нескольких источников (входные данные, которые берутся, например, из очереди сообщений или файловой системы) и заканчивается одним или несколькими приемниками (выходные данные, которые отправляются, например, в очередь сообщений, файловую систему или базу данных). В потоке может быть выполнено произвольное число преобразований. Потоки организованы как ориентированный, ациклический граф, позволяющий приложению распределять и объединять потоки данных. [4]

3.1 Создание контрольных точек

Центральная часть механизма отказоустойчивости Apache Flink — это создание последовательных согласованных контрольных точек потока данных и состояния операторов, к которым система может прибегнуть в случае сбоя. Механизм, используемый для создания этих снимков, базируется на скоординированной контрольной точке и описан в работе «Облегченные асинхронные снимки для распределенных потоков данных». [5] Этот механизм основан на стандартном алгоритме Чанди-Лэмпорта для создания контрольных точек и специально адаптирован к модели исполнения Apache Flink. Важно помнить, что все, что связано с контрольными точками, может выполняться асинхронно.

Основным элементом контрольных точек в Apache Flink являются барьеры потока. Барьеры вставляются в поток данных и передаются вместе с записями как часть потока данных. Барьеры никогда не обгоняют записи, они идут строго последовательно вместе с данными. Барьер разделяет записи в потоке данных на набор записей, которые входят в текущую контрольную точку, и записи, которые принадлежат следующей контрольной точке. Каждый барьер содержит идентификатор контрольной точки, записи соответствующие которой находятся перед барьером. Стоит отметить, что барьеры являются очень легковесными и следовательно не оказывают существенного влияния на течение потока. Несколько барьеров из разных контрольных точек могут находиться в потоке одновременно, как это представлено на рисунке 1. Это означает, что различные контрольные точки могут создаваться одновременно.

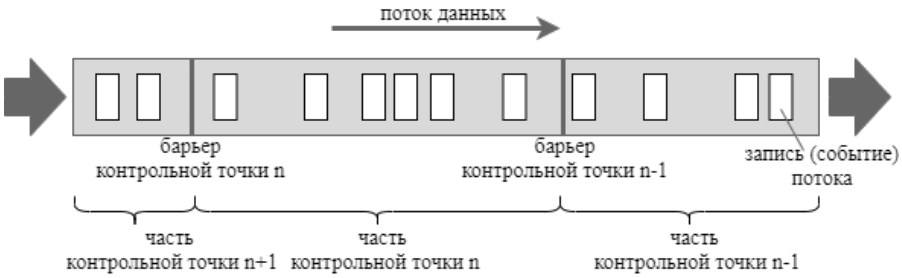

Рисунок 1 – Барьеры контрольных точек

Потоковые барьеры вводятся в поток данных от источников потока. Точка, в которую вставляется барьер для контрольной точки n (назовем ее S_n), — это позиция в исходном потоке, до которой контрольная точка покрывает данные. Например, в Apache Kafka эта позиция будет смещением последней записи в разделе. Эта позиция S_n сообщается координатору контрольной точки.

Затем барьеры проходят вперед по потоку. Когда промежуточный оператор получит барьер n из всех своих входных потоков, он выпустит его во все свои исходящие потоки. Операторы, которые получают более одного входного потока, должны выровнять входные потоки по барьерам контрольных точек.

Как только оператор стока получит барьер n от всех своих входных потоков, он отошлет подтверждение этой контрольной точки n координатору контрольных точек. После того как все стоки подтвердят контрольную точку, она будет считаться завершенной. После завершения контрольной точки n задача больше никогда не будет запрашивать у источника записи, предшествующие S_n, поскольку к этому моменту эти записи уже пройдут через всю топологию потока данных. [6]

3.2 Процесс восстановления после отказа

Процесс восстановления выглядит следующим образом: после сбоя Apache Flink выбирает последнюю пройденную контрольную точку n. Затем система повторно развертывает весь распределенный поток данных и выставляет каждому оператору состояние, которое было сохранено как часть контрольной точки n. Источники настраиваются для начала чтения потока с позиции S_n. Например, в Apache Kafka это означает, что потребитель должен начинать выборку со смещения S_n. Если состояние было создано инкрементально, операторы начинают с состояния последней полной контрольной точки, а затем применяют серию инкрементных обновлений контрольной точки к этому состоянию. [6]

4 Практическая часть

4.1 Описание проблемы

Существующий алгоритм контрольных точек в Apache Flink базируется на асинхронном создании контрольных точек на основе барьеров и имеет очень простой и гибкий протокол, который стабильно работает для заданий, которые можно представить в виде ациклического графа. Алгоритм гарантирует, что все барьеры контрольных точек выровнены в каждом операторе к моменту создания контрольной точки, так что все записи перед барьерами уже обработаны и состояние оператора обновлено в соответствии со строгим порядком следования сообщений. Однако данный алгоритм в текущем его виде неприменим для итеративных вычислений, которые можно представить в виде циклического графа.

Для понимания возможных проблем обратимся к рисунку 2.

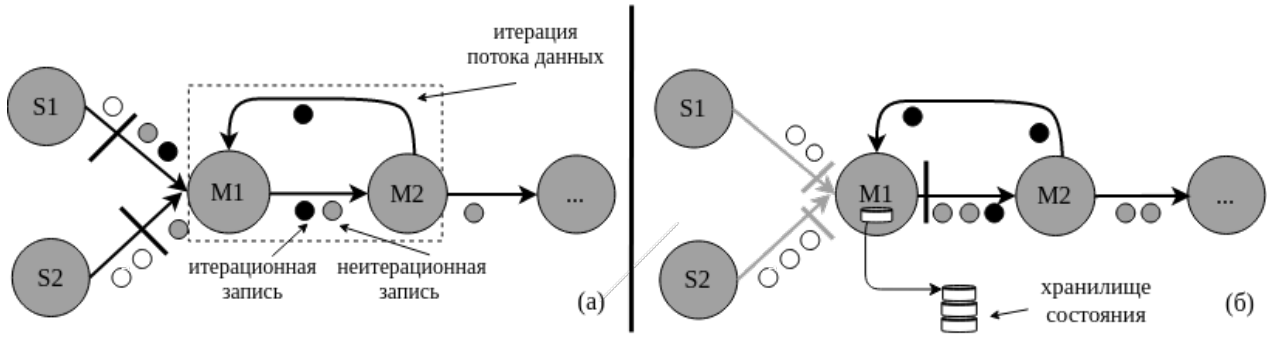

Рисунок 2 – Простая итерация и выравнивание в ней барьеров

На рисунке 2(а) представлена простая итерация в потоке данных с операторами M_1 и M_2. При этом часть записей пройдет через оба оператора строго один раз, а часть записей будет перенаправлена из оператора M_2 в M_1, причем такое зацикливание может быть воспроизведено многократно. На рисунке 2(б) проиллюстрирован момент выравнивания барьеров в операторе M_1, в который должно произойти создание контрольной точки и ее сохранение в хранилище. Очевидно, что на момент создания контрольной точки записи еще не успели пройти через оператор итерации и соответствующим образом обновить его состояние. Таким образом, в хранилище сохраняется некорректное состояние оператора для текущей контрольной точки. В случае отказа системы и восстановления ее по данной контрольной точке, будет использовано некорректное состояние оператора, а также будут потеряны транзитные записи внутри итерации, которые потенциально могли бы быть использованы для восстановления

корректного состояния. По этой причине в настоящие время в Apache Flink по умолчанию отключена возможность использования контрольных точек в итеративном задании. Также уже существует предложение о соответствующем улучшении на официальной странице Apache Flink. [2]

4.2 Предложенное решение

На рисунке 3 пошагово проиллюстрирован способ решения описанной ранее проблемы, основная идея которого приведена в вышеупомянутом предложении об улучшении Apache Flink. Его суть заключается в сохранении в хранилище состояний записей, которые проходят более одного раза через итерацию. Такие записи будем называть итерационными записями.

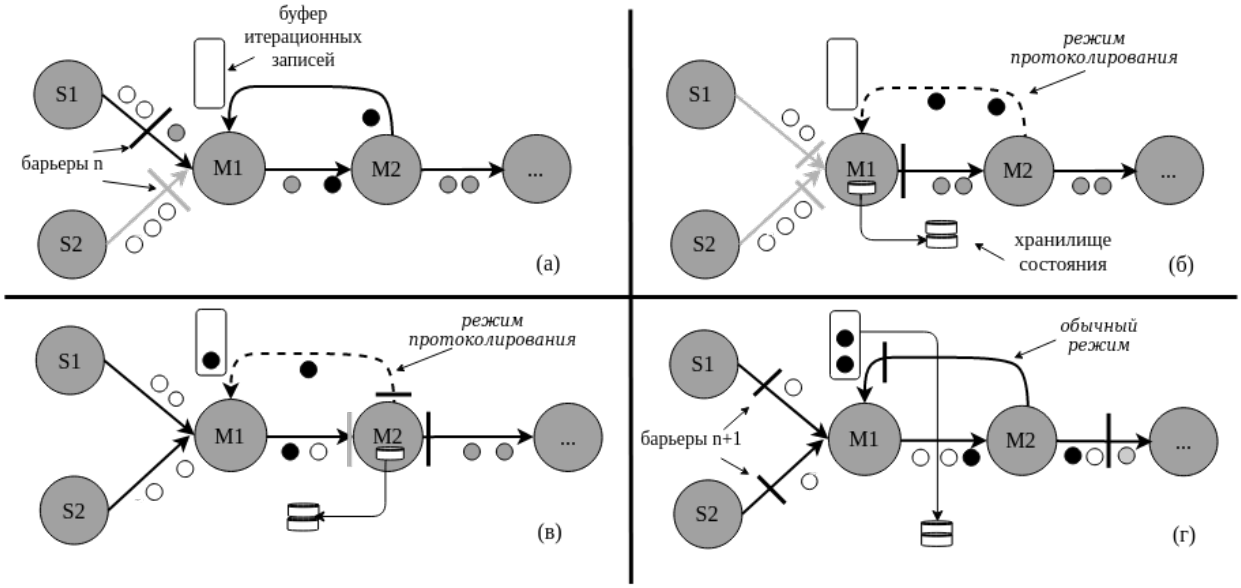

Рисунок 3 – Предложенный алгоритм создания контрольной точки в простой итерации

На рисунке 3(а) представлен поток записей, проходящих через простую итерацию, которая начинается с оператора M_1. Состояние операторов M_1 и M_2 изменяется в результате прохождения через них записей, причем итерационные записи пройдут через цикл итерации более одного раза и соответственно окажут влияние на состояние операторов более одного раза. Стоить отметить, что предложенный алгоритм предполагает сохранение состояний операторов, а также сохранение итерационных записей, проходящих по обратной дуге графа итерации (транзитных сообщений) как части состояния итерации. Для такого сохранения в начало итерации добавляется буфер для итерационных записей. Когда барьеры n из источников S_1 и S_2 выравниваются у входа в оператор M_1, этот оператор сохраняет свое состояние в хранилище состояний, после

чего впускает барьер n внутрь итерации, как проиллюстрировано на рисунке 3(б). Начиная с этого момента обратная дуга между операторами M_2 и M_1 переходит в режим протоколирования, и проходящие по ней итерационные записи начинают сохраняться в буфер, тем самым помечаясь как часть будущей контрольной точки. Как проиллюстрировано на рисунке 3(в), когда барьер n достигает оператора M_2, состояние оператора сохраняется, и барьер раздваивается — один экземпляр отправляется на выход из итерации, другой — в обратную дугу. При достижении барьером n оператора M_1 записи из буфера сохраняются в хранилище состояний как часть контрольной точки n и буфер очищается. После чего обратная дуга переходит в обычный режим работы, что видно на рисунке 3(г). Необходимо заметить, что барьер n из обратной дуги не передается в оператор M_1, а просто отбрасывается. Таким образом, по мере прохождения барьера через тело итерации и обратную дугу, состояния операторов M_1 и M_2, а также транзитные сообщения сохраняются как части контрольной точки n. Когда итерационные записи будут проходить через операторы M_1 и M_2 после прохождения барьера n, то они будут оказывать влияние на состояние операторов. Но так как данные записи уже относятся к контрольной точке $n+1$, то по отношению к ней и барьеру $n+1$ записи проходят через операторы впервые, так как предыдущее их воздействие на операторы уже отражено в контрольной точке n.

Процесс восстановления системы по контрольной точке n после ее отказа выглядит следующим образом. К операторам M_1 и M_2 будут применены соответствующие сохраненные состояния. Из источников данных в поток данных будут направлены записи относящиеся к $n+1$ барьеру, а также на вход итерации, а именно оператора M_1, будут направлены соответствующие сохраненные ранее итерационные записи (транзитные сообщения). Так как данные записи после сохранения контрольной точки n стали относиться к барьеру $n+1$, то при восстановлении системы они также будут отнесены к барьеру $n+1$.

4.3 Детали реализации

Для поддержки итераций в потоковых задачах Apache Flink имеет два специальных типа вершин в графе исполнения: `StreamIterationTail` (хвост итерации) и `StreamIterationHead` (голова итерации).

Хвост итерации неявно добавляет дополнительный оператор `RecordPusher`

в конец тела цикла, который получает все итерационные события от предыдущих операторов и пересылает их голове итерации. Передача данных между хвостом и головой итерации реализована через специальный механизм `BlockingQueueBroker`, который позволяет обмениваться данными между разными системными потоками и гарантирует строгий порядок сообщений. Голова итерации в свою очередь выступает источником данных для начала нового цикла.

Основная идея реализации заключается в том, что после получения барьера голова итерации начинает добавлять проходящие через нее записи в состояние. В свою очередь, помимо данных хвост итерации должен передавать голове итерации барьер обратно. При получении такого барьера голова итерации должна записать сохраненные события в контрольную точку и отбросить барьер. Таким образом, все итерационные записи, проходящие по обратной дуге (транзитные сообщения), гарантированно попадают в контрольную точку.

Алгоритм действий может быть описан следующим образом:

1. Как и раньше голова итерации получает барьер от системы, но теперь:
   1.1. Переходит в режим протоколирования. Это означает, что с этого момента каждая запись, которую она получает от хвоста итерации, добавляется в буфер, который впоследствии попадает в состояние оператора, на шаге 2.
   1.2. Выпускает барьер в нисходящие узлы, что гарантирует завершаемость, а также записи, так как они оказали влияние на состояние операторов, которые станут частью будущей контрольной точки.
2. Как только голова итерации получает барьер обратно от хвоста итерации она:
   2.1. Сохраняет состояние.
   2.2. Очищает буфер с записями.
   2.3. Возвращается к обычному режиму пересылки.
3. При запуске/перезапуске головы итерации, к ней применяется изначальное состояние оператора и сохраненные записи.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы было проведено ознакомление с теоретическими аспектами построения механизмов отказоустойчивости в системах потоковой обработки данных. При этом особое внимание было уделено исследованию подходов к восстановлению состояния систем путем отката. В качестве практического примера отказоустойчивости по контрольным точкам был исследован механизм, существующий в фреймворке Apache Flink. Кроме этого, было реализовано решение проблемы, связанной с сохранением несогласованного состояния системы в контрольной точке для итерационных заданий в нем. Реализация была проверена на корректность с использованием модульного тестирования. В дальнейшем данное решение может быть предложено сообществу Apache Flink для использования в качестве финального решения существующего на сегодняшний день ограничения фреймворка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Клеппман, М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка / М. Клеппман. — СПб.: Питер, 2018. — С. 640.

2 FLIP-16: Loop Fault Tolerance [Электронный ресурс]. — URL: https://cwiki.apache.org/confluence/display/FLINK/FLIP-16%3A+Loop+Fault+Tolerance (Дата обращения 22.04.2020). Загл. с экр. Яз. англ.

3 *Elnozahy, E.* A survey of rollback-recovery protocols in message-passing systems / E. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson // *ACM Computing Surveys*. — 2002. — Vol. 34, no. 3. — Pp. 375–408.

4 Apache Flink [Электронный ресурс]. — URL: https://ru.bmstu.wiki/Apache_Flink (Дата обращения 22.04.2020). Загл. с экр. Яз. рус.

5 *Carbone, P.* Lightweight asynchronous snapshots for distributed dataflows / P. Carbone, G. Fora, S. Ewen, S. Haridi, K. Tzoumas.

6 Apache Flink Documentation [Электронный ресурс]. — URL: https://ci.apache.org/projects/flink/flink-docs-master/ (Дата обращения 22.04.2020). Загл. с экр. Яз. англ.