Arenadata™ Grid

Версия - v2.7

Загрузка и стриминг данных в Arenadata Grid

2020 Arenadata™

Оглавление

1 Загрузка данных **3**
1.1 IgniteDataStreamer . . . 3
1.2 IgniteCache.loadCache() . . . 3
1.3 Загрузка данных Partition-aware . . . 4

2 Стриминг данных **7**
2.1 IgniteDataStreamer . . . 7
2.2 StreamReceiver . . . 8
2.3 StreamTransformer . . . 8
2.4 StreamVisitor . . . 9

В **ADG** возможности загрузки и потоковой передачи (стриминг) данных позволяют принимать в кластер огромные и практически бесконечные объемы данных в масштабируемом и отказоустойчивом режиме. Скорость, с которой данные могут быть введены в **Grid**, очень высока и легко превышает миллионы событий в секунду на кластере среднего размера.

В документации приведено описание загрузки и стриминга данных в Arenadata Grid.

Инструкция может быть полезна администраторам, программистам, разработчикам и сотрудникам подразделений информационных технологий, осуществляющих внедрение системы.

Important: Контактная информация службы поддержки – e-mail: info@arenadata.io

Глава 1

Загрузка данных

ADG предоставляет несколько методов для загрузки данных из сторонней базы данных или другого источника.

Использование стандартных операций кэша *put(. . .)* или *putAll(. . .)* обычно неэффективно для загрузки больших объемов данных. **Grid** предлагает **API** *IgniteDataStreamer*, интеграцию с основными потоковыми технологиями и **API** *IgniteCache*, которые помогают более эффективно загружать большие объемы данных в кластер.

1.1 IgniteDataStreamer

Потоки данных, заданные **API** *IgniteDataStreamer*, предназначены для внедрения больших объемов непрерывных данных в кэш **Grid**. Потоки строятся с возможностью масштабирования и отказоустойчивости и обеспечивают высокую производительность путем группирования записей прежде, чем они будут отправлены соответствующим элементам кластера.

Important: Потоки данных могут использоваться для загрузки большого объема данных в кэш в любое время, включая предварительную загрузку при запуске

1.2 IgniteCache.loadCache()

В случае если сторонняя база данных хранит данные, а приложения используют **SQL**, **Scan** и другие расширенные запросы, **ADG** требует предварительной загрузки данных с диска в оперативную память.

Important: Персистенс ADG не требует прогрева оперативной памяти при перезапуске. Таким образом, методы загрузки, основаные на *IgniteCache.loadCache()*, не актуальны для данного вида персистентного хранения

Для предварительной загрузки данных из стороннего хранилища, такого как реляционная база данных, следует использовать метод *IgniteCache.loadCache()*, позволяющий загружать данные кэша без передачи ключей, которые необходимо загрузить.

Метод *IgniteCache.loadCache()* делегирует метод *CacheStore.loadCache()* для каждого элемента кластера. Чтобы вызвать загрузку только на локальном узле кластера, необходимо использовать метод *IgniteCache.localLoadCache()*.

Important: В случае сегментированных кэшей и персистентности третьей стороны, такой как реляционная база данных, не сопоставленные с этим узлом ключи, так же как их овные и резервные копии, будут автоматически удалены. Это не относится к **Grid Persistent Store**, где каждый узел хранит только те данные, для которых он является основным или резервным

Далее приведен пример использования реализации *CacheStore.loadCache()* для персистентности третьей стороны.

- Java:

```
public class CacheJdbcPersonStore extends CacheStoreAdapter<Long, Person> {
    ...
  // This method is called whenever "IgniteCache.loadCache()" or
  // "IgniteCache.localLoadCache()" methods are called.
  @Override public void loadCache(IgniteBiInClosure<Long, Person> clo, Object... args) {
    if (args == null || args.length == 0 || args[0] == null)
      throw new CacheLoaderException("Expected entry count parameter is not provided.");

    final int entryCnt = (Integer)args[0];

    Connection conn = null;

    try (Connection conn = connection()) {
      try (PreparedStatement st = conn.prepareStatement("select * from PERSONS")) {
        try (ResultSet rs = st.executeQuery()) {
          int cnt = 0;

          while (cnt < entryCnt && rs.next()) {
            Person person = new Person(rs.getLong(1), rs.getString(2), rs.getString(3));

            clo.apply(person.getId(), person);

            cnt++;
          }
        }
      }
    }
    catch (SQLException e) {
      throw new CacheLoaderException("Failed to load values from cache store.", e);
    }
  }
  ...
}
```

1.3 Загрузка данных Partition-aware

В описанном выше скрипте один и тот же запрос выполняется на всех узлах. Каждый узел перебирает весь набор результатов, пропуская ключи, не принадлежащие узлу, что не очень эффективно. Эта ситуация может быть улучшена, если идентификатор раздела хранится рядом с каждой записью в базе данных. Можно использовать интерфейс *org.apache.ignite.cache.affinity.Affinity* для получения идентификатора раздела для любого ключа, хранящегося в кэше.

Далее приведен пример фрагмента кода, который определяет идентификатор раздела для каждого объекта *Person*, хранящегося в кэше.

- Java:

```
IgniteCache cache = ignite.cache(cacheName);
Affinity aff = ignite.affinity(cacheName);

for (int personId = 0; personId < PERSONS_CNT; personId++) {
    // Get partition ID for the key under which person is stored in cache.
    int partId = aff.partition(personId);

    Person person = new Person(personId);
    person.setPartitionId(partId);
    // Fill other fields.

    cache.put(personId, person);
}
```

Когда объекты *Person* становятся зависимыми от идентификаторов раздела, каждый узел может запрашивать только те разделы, которые ему принадлежат. Для этого можно внедрить экземпляр **Grid** в хранилище кэша и использовать его для определения разделов, принадлежащих локальному узлу.

Ниже приведен пример фрагмента кода, демонстрирующий, как использовать *Affinity* для загрузки только локальных разделов. Пример кода является однопоточным, однако его можно эффективно распараллелить идентификатором раздела.

- Java:

```
public class CacheJdbcPersonStore extends CacheStoreAdapter<Long, Person> {
  // Will be automatically injected.
  @IgniteInstanceResource
  private Ignite ignite;

        ...
  // This mehtod is called whenever "IgniteCache.loadCache()" or
  // "IgniteCache.localLoadCache()" methods are called.
  @Override public void loadCache(IgniteBiInClosure<Long, Person> clo, Object... args) {
    Affinity aff = ignite.affinity(cacheName);
    ClusterNode locNode = ignite.cluster().localNode();

    try (Connection conn = connection()) {
      for (int part : aff.primaryPartitions(locNode))
        loadPartition(conn, part, clo);

      for (int part : aff.backupPartitions(locNode))
        loadPartition(conn, part, clo);
    }
  }

  private void loadPartition(Connection conn, int part, IgniteBiInClosure<Long, Person> clo) {
    try (PreparedStatement st = conn.prepareStatement("select * from PERSONS where partId=?")) {
      st.setInt(1, part);

      try (ResultSet rs = st.executeQuery()) {
        while (rs.next()) {
          Person person = new Person(rs.getLong(1), rs.getString(2), rs.getString(3));

          clo.apply(person.getId(), person);
        }
      }
    }
    catch (SQLException e) {
```

```
      throw new CacheLoaderException("Failed to load values from cache store.", e);
    }
  }

  ...
}
```

Сопоставление ключей и разделов зависит от количества разделов, настроенных в *affinity* функции (*org.apache.ignite.cache.affinity.AffinityFunction*). При изменении ее конфигурации записи идентификаторов разделов в базе данных должны быть обновлены соответствующим образом.

Important: Для соблюдения согласованности и долговечности Grid persistence поддерживает запись в формате Write-Ahead, включенной по умолчанию. Однако, это может повлиять на производительность кластера во время предварительной загрузки данных. Рекомендуется отключать WAL при предварительной загрузке данных и включать после ее завершения

Глава 2

Стриминг данных

Как работает стриминг:

1. Клиентские узлы вводят конечные или непрерывные потоки данных в кэш **Grid**, используя **Grid Data Streamers**.
2. Данные автоматически разделяются между узлами **Grid**, и каждый узел получает равное количество данных.
3. Потоковые данные могут совместно и одновременно обрабатываться на узлах данных **Grid**.
4. Клиенты также могут выполнять параллельные SQL-запросы для потоковых данных.

Потоки данных определяются **API** *IgniteDataStreamer* и создаются для ввода больших объемов непрерывных потоков данных в кэш **Grid**. Потоки данных построены с возможностью масштабирования и отказоустойчивости и обеспечивают семантику по крайней мере один раз для всех данных, передаваемых в **ADG**.

Потоки данных не участвуют в транзакциях **Grid**.

Для запроса потоковых данных можно использовать полный набор возможностей индексирования данных в **Grid**, вместе с кэшированными запросами **SQL**, **TEXT** и **Predicate**.

ADG интегрируется с основными потоковыми технологиями и платформами, такими как **Kafka**, **Camel**, **Storm** или **JMS**, чтобы обеспечить еще более продвинутые возможности потоковой передачи на архитектуре **Grid**.

2.1 IgniteDataStreamer

Основной абстракцией для быстрой потоковой передачи больших объемов данных в **Grid** является *IgniteDataStreamer*, который внутренне правильно группирует ключи вместе и совмещает эти пакеты с узлами, на которых данные будут кэшироваться.

Высокая скорость загрузки достигается благодаря следующим методам:

- Записи, сопоставленные одному и тому же элементу кластера, объединяются в буфер;
- Несколько буферов могут сосуществовать одновременно;
- Чтобы избежать нехватки памяти, Data streamer имеет максимальное количество буферов, которые он может обрабатывать одновременно.

  Для добавления данных в поток необходимо вызвать метод *IgniteDataStreamer.addData(. . .)*.

- Java:

```
// Get the data streamer reference and stream data.
try (IgniteDataStreamer<Integer, String> stmr = ignite.dataStreamer("myStreamCache")) {
    // Stream entries.
    for (int i = 0; i < 100000; i++)
        stmr.addData(i, Integer.toString(i));
}
```

По умолчанию стример данных не перезаписывает существующие данные, а это означает, что если он встречает запись, которая уже находится в кэше, то он пропускает ее. Это наиболее эффективный режим работы, так как потоку данных не нужно беспокоиться об управлении версиями данных в фоновом режиме.

В случае если данные могут уже находиться в потоковом кэше, но при этом необходимо их перезаписать, следует установить параметр *DataStreamer.allowOverwrite(true)*.

Important: Свойство *AllowOverwrite* при значении *false* (по умолчанию) также пропускает Persistent Store, даже если в свойстве *SkipStore* установлено *false*. Хранилище кэша вызывается только тогда, когда *AllowOverwrite* имеет значение *true*

2.2 StreamReceiver

Для случаев, когда необходимо выполнить какую-то пользовательскую логику, а не при добавлении новых данных, можно воспользоваться **API** *StreamReceiver*.

Потоковые приемники позволяют реагировать на потоковые данные в сочетании друг с другом непосредственно на узлах, где они будут кэшироваться. Данные можно изменить или добавить к ним какую-либо пользовательскую логику предварительной обработки прежде, чем помещать их в кэш.

Important: *StreamReceiver* не помещает данные в кэш автоматически. Для этого необходимо воспользоваться любым из методов *cache.put(...)*

2.3 StreamTransformer

StreamTransformer – это удобная реализация *StreamReceiver*, которая обновляет данные в потоке кэша на основе его предыдущего значения. Обновление совмещенное, то есть оно происходит точно на узле кластера, где хранятся данные.

В приведенном примере используется *StreamTransformer* для увеличения счетчика для каждого отдельного слова, найденного в текстовом потоке.

- transformer:

```
CacheConfiguration cfg = new CacheConfiguration("wordCountCache");

IgniteCache<String, Long> stmCache = ignite.getOrCreateCache(cfg);

try (IgniteDataStreamer<String, Long> stmr = ignite.dataStreamer(stmCache.getName())) {
  // Allow data updates.
  stmr.allowOverwrite(true);

  // Configure data transformation to count instances of the same word.
  stmr.receiver(StreamTransformer.from((e, arg) -> {
    // Get current count.
```

```
    Long val = e.getValue();

    // Increment count by 1.
    e.setValue(val == null ? 1L : val + 1);

    return null;
  }));

  // Stream words into the streamer cache.
  for (String word : text)
    stmr.addData(word, 1L);
}
```

- java7 transformer:

```
CacheConfiguration cfg = new CacheConfiguration("wordCountCache");

IgniteCache<Integer, Long> stmCache = ignite.getOrCreateCache(cfg);

try (IgniteDataStreamer<String, Long> stmr = ignite.dataStreamer(stmCache.getName())) {
  // Allow data updates.
  stmr.allowOverwrite(true);

  // Configure data transformation to count instances of the same word.
  stmr.receiver(new StreamTransformer<String, Long>() {
    @Override public Object process(MutableEntry<String, Long> e, Object... args) {
      // Get current count.
      Long val = e.getValue();

      // Increment count by 1.
      e.setValue(val == null ? 1L : val + 1);

      return null;
    }
  });

  // Stream words into the streamer cache.
  for (String word : text)
    stmr.addData(word, 1L);
```

2.4 StreamVisitor

StreamVisitor также является удобной реализацией *StreamReceiver*, которая просматривает каждую запись ключ-значение в потоке. При этом пользователь не обновляет кэш. Если запись необходимо сохранить в кэше, следует вызвать любой из методов *cache.put(...)*.

В приведенном примере имеется 2 кэша: “marketData” и “instruments”. При получении меток рыночных данных, они помещаются в стример для кэша “marketData”. *StreamVisitor* для “marketData” вызывается для элемента кластера, сопоставленного с конкретным обозначением рынка. В этот момент обновляется кэш “instrument” с последней рыночной ценой. При этом кэш “marketData” не обновляется вообще, оставаясь пустым. Он просто используется для совмещенной обработки рыночных данных в кластере непосредственно на том узле, где будут храниться данные.

- Java:

```
CacheConfiguration<String, Double> mrktDataCfg = new CacheConfiguration<>("marketData");
CacheConfiguration<String, Double> instCfg = new CacheConfiguration<>("instruments");
```

```
// Cache for market data ticks streamed into the system.
IgniteCache<String, Double> mrktData = ignite.getOrCreateCache(mrktDataCfg);

// Cache for financial instruments.
IgniteCache<String, Double> instCache = ignite.getOrCreateCache(instCfg);

try (IgniteDataStreamer<String, Integer> mktStmr = ignite.dataStreamer("marketData")) {
  // Note that we do not populate 'marketData' cache (it remains empty).
  // Instead we update the 'instruments' cache based on the latest market price.
  mktStmr.receiver(StreamVisitor.from((cache, e) -> {
    String symbol = e.getKey();
    Double tick = e.getValue();

    Instrument inst = instCache.get(symbol);

    if (inst == null)
      inst = new Instrument(symbol);

    // Update instrument price based on the latest market tick.
    inst.setHigh(Math.max(inst.getLatest(), tick);
    inst.setLow(Math.min(inst.getLatest(), tick);
    inst.setLatest(tick);

    // Update instrument cache.
    instCache.put(symbol, inst);
  }));

  // Stream market data into Ignite.
  for (Map.Entry<String, Double> tick : marketData)
      mktStmr.addData(tick);
}
```