Arenadata™ Grid

Версия - v2.4

Durable Memory в Arenadata Grid

2020 Arenadata™

Оглавление

1 Durable Memory **3**
1.1 Функции In-Memory . . . 3
1.2 Функции персистентности . . . 3

2 Архитектура памяти **5**
2.1 Область данных . . . 5
2.2 Оперативные и исторические данные . . . 5
2.3 Сегмент памяти . . . 6
2.4 Страницы данных . . . 6
2.5 B+ деревья и индексные страницы . . . 6
2.6 Свободные списки . . . 7

3 Конфигурация памяти **8**
3.1 Общие параметры конфигурации . . . 9
3.2 Области данных . . . 11
3.3 Кэширование On-heap . . . 14

В документации приведено описание памяти Durable Memory, ее архитектура и конфигурация.

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

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

Глава 1

Durable Memory

Ориентированная на память платформа **Arenadata Grid** основана на долговременной архитектуре памяти, которая позволяет хранить и обрабатывать данные и индексы как в памяти, так и на диске (при включенной функции *Grid Native Persistence*). Архитектура Durable Memory помогает достичь функционирование памяти эффективней дисковой, используя все доступные ресурсы кластера.

Долговременная память **Grid** работает аналогично виртуальной памяти операционных систем (таких как **Linux**). Однако, одно существенное различие между этими двумя типами архитектуры заключается в том, что долговременная память при включенной персистентности всегда обрабатывает диск как надмножество данных, способное выдержать сбои и перезагрузки, в то время как традиционная виртуальная память использует диск только в качестве пространства подкачки памяти, которое очищается после завершения процесса.

1.1 Функции In-Memory

Поскольку архитектура ориентирована на память, оперативная память всегда рассматривается как первый уровень, где происходит вся обработка. Далее приведены характеристики и преимущества архитектуры в части in-memory:

- **Off-heap Based** – все данные и индексы хранятся вне **Java heap**, что позволяет легко обрабатывать петабайты данных, находящихся в кластере;
- **Removes Noticeable Garbage Collection Pauses** – поскольку все данные и индексы хранятся off-heap, код приложения является единственным возможным источником для длительных пауз;
- **Предсказуемое использование памяти** – по умолчанию долговременная память использует как можно больше RAM и дискового пространства. Тем не менее, можно легко настроить использование памяти в соответствии с необходимыми требованиями приложения;
- **Автоматическая дефрагментация памяти** – **Arenadata Grid** использует память как можно более эффективно и выполняет процедуры дефрагментации в фоновом режиме, избегая фрагментации;
- **Повышенная производительность и использование памяти** – все данные и индексы хранятся в постраничном формате с аналогичным представлением в памяти и на диске, что устраняет необходимость сериализации или десериализации данных.

1.2 Функции персистентности

В данном разделе описываются параметры персистентности в **Grid**:

- *Grid Native Persistence*

- *Сторонняя персистентность*
- *Пространство подкачки памяти*

1.2.1 Grid Native Persistence

Grid Persistence – это наиболее гибкий, масштабируемый и удобный способ хранения данных в **ADG**. Он широко используется в сценариях, где приложениям требуется распределенная база данных, ориентированная на память. Преимущества следующие:

- **Устойчивость Данных**. Поскольку на уровне *Grid persistent* хранится полный набор данных, он может выдержать сбои и перезапуски кластера без потери данных и с сохранением высокой устойчивости транзакций;
- **Кэш только горячих данных в памяти**. Хранение надмножества данных на диске и подмножества данных в памяти. При включенной функции *Grid Native Persistence* нет необходимости помещать все данные в RAM. Долговременная память сохраняет горячие данные в RAM, автоматически очищая холодные данные из памяти, когда больше не остается места;
- **Выполнение SQL по всем наборам данных**. Большинство in-memory систем могут запрашивать данные только в том случае, когда они предварительно загружены в память, поэтому ограничение размера данных зависит только от размера памяти. **Grid SQL** всегда охватывает весь распределенный персистентный набор данных, используя кэш in-memory для повышения производительности;
- **Мгновенный перезапуск кластера**. Если весь кластер выходит из строя, он может перезагрузиться и сразу же начать работу. In-memory кэш автоматически разогревается по мере доступа к данным.

1.2.2 Сторонняя персистентность

ADG может использоваться как слой кэширования (сетка данных) над сторонней базой данных – **RDBMS**, **NoSQL** или **HDFS**. Этот режим используется для ускорения основной БД, в которой сохраняются данные. **Grid** хранит данные в памяти, распределенной по нескольким узлам, обеспечивая быстрый доступ к данным. Это снижает нагрузку на сеть, вызванную частым перемещением данных между приложением и БД.

Однако существуют некоторые ограничения по сравнению с собственной персистентностью. Например, SQL-запросы выполняются только по данным, находящимся в оперативной памяти, что требует предварительную загрузку всего набора данных с диска в память.

1.2.3 Пространство подкачки памяти

Вместо использования персистентности **Grid** или сторонней персистентности, можно включить swapping (подкачку), и в таком случае при исчерпанной памяти **RAM** данные из in-memory **Grid** перемещаются в пространство подкачки, расположенное на диске. При включенном пространстве подкачки (swap space), **ADG** хранит данные в файлах с отображением в памяти (memory mapped files, **MMF**), содержимое которых может перемещаться с ОС на диск в зависимости от текущего потребления **RAM**.

Swap space в основном используется во избежание ошибок в памяти (out of memory errors, **OOME**), которые могут произойти, если потребление оперативной памяти превышает ее емкость, потребуется больше времени для масштабирования кластера с целью равномерного распределения наборов данных.

Глава 2

Архитектура памяти

Grid Durable Memory – это архитектура памяти на постраничной основе, разбитая на страницы фиксированного размера. Страницы хранятся в управляемых областях off-heap в **RAM** (вне **Java heap**) и организованы в особой иерархии на диске.

Grid поддерживает то же представление двоичных данных как в памяти, так и на диске, что устраняет необходимость дорогостоящей сериализации при перемещении данных между памятью и диском.

- *Область данных*
- *Оперативные и исторические данные*
- *Сегмент памяти*
- *Страницы данных*
- *B+ деревья и индексные страницы*
- *Свободные списки*

2.1 Область данных

Область данных – это логически расширяемая область. Долговременная память может состоять из одной или нескольких областей данных, которые могут различаться по размеру и политике, персистироваться на диск и иметь другие отличия.

По умолчанию **Grid** выделяет одну область данных, занимающую до *20%* оперативной памяти, доступной на локальном узле кластера.

2.2 Оперативные и исторические данные

Для лучшей производительности следует хранить оперативные данные в памяти. Для этого необходимо настроить несколько областей памяти.

Например, предположим, что имеются объекты *Person*, *Purchases* и *Records*, хранящиеся в *PersonCache*, *PurchasesCache* и *RecordsCache* соответственно. Данные о лицах и закупках являются оперативными, то есть данными, к которым часто обращаются. А данные записей – исторические, доступ к ним осуществляется реже.

Теперь предположим, что есть только *200 ГБ* оперативной памяти. В этом случае физическую память можно разделить следующим образом:

- *190 ГБ* на создание области для оперативных или часто используемых данных, таких как *Person* и *Purchases*. Таким образом, *PersonCache* и *PurchaseCache* получают максимальную производительность из кластера;
- *10 ГБ* на выделение области для исторических или редко используемых наборов данных, таких как *RecordsCache*, данные которого размещаются на диске.

2.3 Сегмент памяти

Каждая область данных начинается с исходного размера и имеет максимальный размер, до которого может вырасти. Область расширяется до максимальной границы, выделяя сегменты непрерывной памяти. По умолчанию максимальный размер области равен 20% объема **RAM**, доступного в системе.

Сегмент памяти представляет собой непрерывный массив байтов или физическую память, выделенную из операционной системы. Массив разделяется на страницы фиксированного размера. Существует несколько типов страниц, которые могут находиться в сегменте.

2.4 Страницы данных

Страница данных хранит записи, помещенные в кэш **ADG** со стороны приложения.

Как правило, одна страница данных содержит несколько записей ключ-значение для того, чтобы использовать память как можно более эффективно и избежать фрагментации памяти. Когда новая запись добавляется в кэш, **Grid** выполняет поиск оптимальной страницы, которая может соответствовать всей записи ключ-значение.

Однако, если общий размер записи превышает размер страницы, настроенный с помощью параметра *DataStorageConfiguration.setPageSize(..)*, то запись занимает более одной страницы данных.

2.5 B+ деревья и индексные страницы

Все индексы **SQL**, определенные и используемые в приложении, поддерживаются в древовидной структуре данных *B+*. Для каждого уникального индекса в схеме **SQL**, **Grid** создает и управляет выделенным экземпляром дерева *B+*.

Кэшированные ключи также хранятся в деревьях *B+* и упорядочены по их значениям хэш-кода.

Цель дерева *B+* состоит в том, чтобы связать и упорядочить индексные страницы, которые выделяются и хранятся в долговременной памяти. В индексной странице содержится вся информация, необходимая для определения индексированного значения и перемещения записи на странице данных, а также ссылки на другие индексные страницы дерева.

B+ tree Meta Page необходима, чтобы добраться до корня конкретного дерева *B+* и его слоев, для эффективного выполнения ряда запросов. Например, при выполнении операции *myCache.get(keyA)*, вызывается следующий поток действий:

1. **Grid** выполняет поиск области памяти, к которой принадлежит *myCache*;
2. Размещается мета-страница, указывающая на хэш-индекс *B+* дерева *myCache*;
3. На основе хэш-кода *keyA* индексная страница, к которой принадлежит ключ, располагается в дереве *B+*;
4. Если соответствующая индексная страница не найдена в памяти или на диске, **Grid** делает вывод о том, что ключ не существует, и выдает значение *null*;
5. Если индексная страница существует, то она содержит всю информацию, необходимую для поиска страницы данных кэша *keyA*;

6. **Grid** находит страницу данных для *keyA* и выдает значение пользователю.

2.6 Свободные списки

Вышеуказанный поток выполнения объясняет, как кэш просматривается в памяти страницы. Теперь перейдем к тому, как **Grid** сохраняет новую запись кэша при выполнении такой операции, как *myCache.put (keyA, valueA)*.

В этом случае долговременная память основывается на структуре свободных списков. Свободный список – это двунаправленный список, в котором хранятся ссылки на страницы памяти примерно равного свободного места. Страницы данных и индексов отслеживаются в разных свободных списках.

Далее приведен поток выполнения операции *myCache.put (keyA, valueA)*:

1. **Grid** выполняет поиск области памяти, к которой принадлежит *myCache*;
2. Размещается мета-страница, указывающая на хэш-индекс *B+* дерева *myCache*;
3. На основе хэш-кода *keyA* индексная страница, к которой принадлежит ключ, располагается в дереве *B+*;
4. В случае если соответствующая индексная страница не найдена в памяти или на диске, запрашивается новая страница у одного из свободных списков. Как только индексная страница будет предоставлена, она добавляется в дерево *B+*;
5. Если индексная страница пуста (т.е. не относится ни к одной странице данных), то страница данных предоставляется одним из свободных списков в зависимости от общего размера записи кэша. Ссылка на страницу данных добавляется на индексную страницу;
6. Запись кэша добавляется на страницу данных.

Глава 3

Конфигурация памяти

По умолчанию узлы **Grid** потребляют до *20%* оперативной памяти, доступной локально, и в большинстве случаев это единственный параметр, который может потребоваться изменить. Для этого следует внести правки в размер области данных:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">

<!-- Redefining maximum memory size for the cluster node usage. -->
<property name="dataStorageConfiguration">
  <bean class="org.apache.ignite.configuration.DataStorageConfiguration">
    <!-- Redefining the default region's settings -->
    <property name="defaultDataRegionConfiguration">
      <bean class="org.apache.ignite.configuration.DataRegionConfiguration">
        <property name="name" value="Default_Region"/>
        <!-- Setting the size of the default region to 4GB. -->
        <property name="maxSize" value="#{4L * 1024 * 1024 * 1024}"/>
      </bean>
    </property>
  </bean>
</property>

<!-- The rest of the parameters. -->
</bean>
```

- Java:

```
IgniteConfiguration cfg = new IgniteConfiguration();

// Changing total RAM size to be used by Ignite Node.
DataStorageConfiguration storageCfg = new DataStorageConfiguration();

// Setting the size of the default memory region to 4GB to achieve this.
storageCfg.getDefaultDataRegionConfiguration().setMaxSize(
    4L * 1024 * 1024 * 1024);

cfg.setDataStorageConfiguration(storageCfg);

// Starting the node.
Ignition.start(cfg);
```

Подробные настройки долговременной памяти приведены в следующих разделах:

- *Общие параметры конфигурации*
- *Области данных*
- *Кэширование On-heap*

3.1 Общие параметры конфигурации

Для изменения основных настроек долговременной памяти, таких как размер страницы, необходимо использовать *org.apache.ignite.configuration.DataStorageConfiguration* с помощью метода *IgniteConfiguration.setDataStorageConfiguration(...)*. В таблице представлены некоторые доступные параметры.

Таблица3.1.: Параметры конфигурации Durable Memory

Параметр	Описание	Значение по умолчанию
setPageSize(...)	Устанавливает размер страницы по умолчанию. Как правило, параметр правится, когда приложение генерирует много объектов, которые не вписываются в одну страницу	4 KB
setDefaultDataRegionConfiguration(...)	Задает размер области данных, создающейся автоматически, по умолчанию. Если свойство не задано, то область может использовать до 20% оперативной памяти, доступной на локальном компьютере	20% от RAM; персистентность отключена
setDataRegionConfigurations(...)	Устанавливает список всех областей данных, настроенных на узле кластера (*Области данных*)	Пустой массив. Конфигурация, используемая для создания области по умолчанию, здесь не хранится
setSystemRegionInitialSize(...)	Устанавливает начальный размер области данных, предназначенной для системных запросов	40 MB
setSystemRegionMaxSize(...)	Устанавливает максимальный размер области данных, предназначенной для системных запросов. Не менее 10 МБ (из-за накладных расходов внутренних данных)	100 MB
setConcurrencyLevel(...)	Устанавливает количество параллельных сегментов в таблицах сопоставления внутренних страниц ADG	Общее количество доступных процессоров, умноженное на 4

В следующем примере показано, как изменить размер страницы и уровень параллелизма с помощью *DataStorageConfiguration*:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
  <property name="dataStorageConfiguration">
    <bean class="org.apache.ignite.configuration.DataStorageConfiguration">
      <!-- Set concurrency level -->
      <property name="concurrencyLevel" value="4"/>

      <!-- Set the page size to 8 KB -->
      <property name="pageSize" value="8192"/>
    </bean>
  </property>

  <!--- Additional settings ---->
</bean>
```

- Java:

```
// Ignite configuration.
IgniteConfiguration cfg = new IgniteConfiguration();

// Durable memory configuration.
DataStorageConfiguration storageCfg = new DataStorageConfiguration();

// Altering the concurrency level.
storageCfg.setConcurrencyLevel(4);

// Changing the page size to 8 KB.
storageCfg.setPageSize(8192);

// Applying the new configuration.
cfg.setDataStorageConfiguration(storageCfg);
```

3.2 Области данных

По умолчанию долговременная память создает единую расширяемую область данных, которая может занимать до *20%* оперативной памяти, доступной на локальном компьютере, с отключенным персистенсом. Для определения нескольких областей данных с различными параметрами, такими как размер области, персистенс или политика вытеснения, используется класс *org.apache.ignite.configuration.DataRegionConfiguration*.

Например, для создания области данных, потребляющей до *500 МБ* оперативной памяти и с включенным персистенсом, конфигурацию необходимо задать следующим образом:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
  <!-- Durable memory configuration. -->
  <property name="dataStorageConfiguration">
    <bean class="org.apache.ignite.configuration.DataStorageConfiguration">
      <property name="dataRegionConfigurations">
        <list>
          <!--
              Defining a data region that will consume up to 500 MB of RAM and
              will have eviction and persistence enabled.
          -->
          <bean class="org.apache.ignite.configuration.DataRegionConfiguration">
            <!-- Custom region name. -->
            <property name="name" value="500MB_Region"/>
```

```
            <!-- 100 MB initial size. -->
            <property name="initialSize" value="#{100L * 1024 * 1024}"/>

            <!-- 500 MB maximum size. -->
            <property name="maxSize" value="#{500L * 1024 * 1024}"/>

            <!-- Enabling persistence for the region. -->
            <property name="persistenceEnabled" value="true"/>
          </bean>
        </list>
      </property>
    </bean>
  </property>

  <!-- Other configurations. -->
</bean>
```

- Java:

```
// Ignite configuration.
IgniteConfiguration cfg = new IgniteConfiguration();

// Durable Memory configuration.
DataStorageConfiguration storageCfg = new DataStorageConfiguration();

// Creating a new data region.
DataRegionConfiguration regionCfg = new DataRegionConfiguration();

// Region name.
regionCfg.setName("500MB_Region");

// Setting initial RAM size.
regionCfg.setInitialSize(100L * 1024 * 1024);

// Setting maximum RAM size.
regionCfg.setMaxSize(500L * 1024 * 1024);

// Enable persistence for the region.
regionCfg.setPersistenceEnabled(true);

// Setting the data region configuration.
storageCfg.setDataRegionConfigurations(regionCfg);

// Applying the new configuration.
cfg.setDataStorageConfiguration(storageCfg);
```

Затем, чтобы использовать настроенную область, необходимо поручить **Grid caches** хранить там данные, передав имя области (*500MB_Region*) в метод *org.apache.ignite.configuration.CacheConfiguration.setDataRegionName(...)*:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    <!-- Durable Memory and other configuration parameters. -->
    <!-- ....... -->

    <property name="cacheConfiguration">
       <list>
```

```
            <!-- Cache that is mapped to a specific data region. -->
            <bean class="org.apache.ignite.configuration.CacheConfiguration">
               <!--
                    Assigning the cache to the `500MB_Region` defined earlier.
                -->
                <property name="dataRegionName" value="500MB_Region"/>

                      <!-- Cache name. -->
                <property name="name" value="SampleCache"/>

                <!-- Additional cache configuration parameters -->
            </bean>
        </list>
    </property>

    <!-- The rest of the configuration. -->
    <!-- ....... -->
</bean>
```

- Java:

```
// Ignite configuration.
IgniteConfiguration cfg = new IgniteConfiguration();

// Durable Memory configuration and the rest of the configuration.
// ....

// Creating a cache configuration.
CacheConfiguration cacheCfg = new CacheConfiguration();

// Binding the cache to the earlier defined region.
cacheCfg.setDataRegionName("500MB_Region");

// Setting the cache name.
cacheCfg.setName("SampleCache");

// Applying the cache configuration.
cfg.setCacheConfiguration(cacheCfg);
```

Как только узел кластера запускается с настроенной конфигурацией, долговременная память выделяет область данных, изначально потребляющую *100 МБ* **RAM**, и которая при этом может вырасти до *500 МБ*. Надмножество данных всегда хранится на диске, гарантируя, что потеря данных не произойдет, даже если в оперативной памяти больше не останется свободного места. **Grid** автоматически вытесняет наименее недавно используемые данные из **RAM** при включенном персистенсе. Заданная область хранит данные, добавленные только в *SampleCache*. Остальные кэши направляются к области данных по умолчанию, если не указана иная.

В случае если персистенс **Grid** отключен и общее использование **RAM** превышает *500 МБ*, происходит изъятие из памяти. Во избежение этого необходимо:

- Использовать персистенс, как показано в примере выше;
- Использовать один из доступных алгоритмов вытеснения. Необходимо знать, что вытеснение включено по умолчанию, если используется персистенс;
- Установить максимальный размер области данных на большее значение.

Полный список параметров *DataRegionConfiguration* приведен в javadoc.

3.3 Кэширование On-heap

Grid Durable Memory – это off-heap память, которая распределяет все области памяти за пределы **Java heap**. Но при этом можно включить on-heap кэширование, установив для параметра *org.apache.ignite.configuration.CacheConfiguration.setOnheapCacheEnabled(...)* значение *true*.

On-heap кэширование полезно для сценариев, когда выполняется много операций чтения кэша на узлах сервера, которые работают с записями кэша в бинарной форме или вызывают десериализацию записей кэша. К примеру, это может произойти, когда распределенный вычислительный или развернутый сервис получает некоторые данные из кэша для дальнейшей обработки.

Important: Для управления размером on-heap кэша и избежания его постоянного роста должна быть включена одна из доступных записей кэша на основе политики вытеснения