Arenadata™ Grid

Версия - v2.3

Руководство по работе с Arenadata Grid

2018 Arenadata™

Оглавление

1 Основные концепции Arenadata Grid **3**
1.1 Факты об Arenadata Grid . . . 3
1.2 Клиенты и серверы . . . 4

2 Durable Memory **10**
2.1 Durable Memory . . . 10
2.2 Архитектура памяти . . . 12
2.3 Конфигурация памяти . . . 14

3 Хранилище “ключ-значение” **21**
3.1 Data Grid . . . 21
3.2 Запросы к данным в кэше . . . 22
3.3 Режимы кэша . . . 26
3.4 Основные и резервные копии . . . 27

4 Загрузка и стриминг данных **30**
4.1 Загрузка данных . . . 30
4.2 Стриминг данных . . . 33

В документе приведено общее описание Arenadata Grid: основные концепции, Durable Memory и хранилище “ключ-значение”, загрузка и стриминг данных.

Документ может быть полезен администраторам, программистам, разработчикам и сотрудникам подразделений информационных технологий, осуществляющих внедрение и сопровождение кластеров Arenadata.

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

Глава 1

Основные концепции Arenadata Grid

Платформа базы данных, ориентированная на память, кэширование и обработку данных

Arenadata Grid (**ADG**) – это ориентированная на память распределенная база данных для транзакционных, аналитических и потоковых рабочих нагрузок, обеспечивающая скорость в памяти в масштабе петабайтов.

- *Факты об Arenadata Grid*
- *Клиенты и серверы*

1.1 Факты об Arenadata Grid

- Хранит ли Grid данные на диске, или только в памяти?

И то и другое. Локальное сохранение данных на диск в **Grid** может быть включено и выключено. Это позволяет платформе хранить больший объём данных, чем может поместиться в памяти. По сути, небольшой объём оперативных данных может быть размещён в памяти, а больший набор данных, который не вписывается в память, может храниться на диске, используя память в качестве кеширующего слоя для лучшей производительности.

- Является ли Grid in-memory database (IMDB)?

Да. Несмотря на то, что **Grid** с успехом хранит данные как в памяти, так и на диске, использование диска может быть отключено, а **Grid** будет выступать в качестве распределенной базы данных в памяти, с поддержкой **SQL** и распределенных транзакций.

- Является ли Grid in-memory data grid (IMDG)?

Да. **Grid** – это полнофункциональное распределённое хранилище данных, которое может использоваться либо в режиме только памяти, либо с использованием памяти и дискового накопителя. **Grid** также может автоматически интегрироваться с любыми сторонними базами данных, включая любые СУБД или хранилища **NoSQL**.

- Является ли Grid распределенным кэшем?

Да. Если использование диска отклчено, **Grid** становится распределенным кэшем. **Grid** реализует спецификацию **JCache** (**JSR 107**) и обеспечивает большую функциональность, чем требуется спецификацией, включая режимы раздельного и реплицированного распространения, распределённые транзакции **ACID**, SQL-запросы, хранение на диске и т.д.

- Является ли Grid распределенной базой данных?

Да. Данные в **Grid** секционируются или реплицируются в кластере из нескольких узлов. Это обеспечивает масштабируемость и повышает устойчивость системы. **Grid** автоматически контролирует разделение данных, однако, пользователи могут подключать свои собственные функции распределения (affinity) и совместно использовать разные части данных для повышения эффективности.

- Является ли Grid базой данных SQL?

Не в полной мере. Хотя **Grid** стремится вести себя как любая другая реляционная база данных **SQL**, существуют различия в том, как **Grid** обрабатывает ограничения и индексы. **Grid** поддерживает первичные и вторичные индексы, однако уникальность может быть применена только для первичных индексов. **Grid** также не поддерживает ограничения внешнего ключа.

По сути, **Grid** намеренно не поддерживает никаких ограничений, которые повлекли бы за собой рассылку сообщений о каждом обновлении кластера и значительно ухудшают производительность и масштабируемость системы.

- Является ли Grid базой данных NoSQL?

Не совсем. Как и другие базы данных **NoSQL**, **Grid** высокодоступен и масштабируется по горизонтали. Однако, в отличие от других баз данных **NoSQL**, **Grid** поддерживает транзакции **SQL** и **ACID**.

- Является ли Grid транзакционной базой данных?

Не в полной мере. Транзакции **ACID** поддерживаются, но только на уровне **API** с ключом. **Grid** также поддерживает транзакции между разделами, что означает, что транзакции могут охватывать ключи, находящиеся в разных разделах на разных серверах.

На уровне **SQL Grid** поддерживает атомарную, но еще не транзакционную согласованность. Реализация SQL-транзакции планируется в версии *2.4*.

- Является ли Grid основой для нескольких моделей?

Да. **Grid** поддерживает как ключ-значение, так и **SQL** для моделирования и доступа к данным. Кроме того, **Grid** предоставляет надежную обработку **API** для вычисления распределенных данных.

- Является ли Grid хранилищем ключ-значений?

Да. **Grid** предоставляет многофункциональный ключ-значение **API**, который совместим с **JCache** (**JSR-107**) и поддерживает **Java**, **C++** и **.NET**.

- Что такое долговременная память (durable memory)?

Архитектура долговременной памяти позволяет **Grid** использовать диск как расширяющее основную память хранилище. По умолчанию данные хранялся в памяти вне кучи (off-heap). По мере вытеснения или обновления существующих данных, изменения попадают сначала в журнал записи (**WAL** - Write-ahead log), а затем в основное постоянное хранилище **Grid**. В случае, если использование диска отключено, долговременная память действует как простое хранилище.

- Что такое совмещенная обработка?

Grid – это распределенная система, и поэтому важно правильно распределять данные по узлам кластера, а также максимально совмещать данные с алгоритмами их обработки, чтобы избежать затратных перемещений между узлами. Совмещенная обработка данных становится особенно важной при выполнении распределенных SQL-запросов, таких как join. **Grid** также поддерживает отправку пользовательской логики (функции, лямбды и т.д.) непосредственно к узлам, где данные хранятся и обрабатываются, достигая таким обрабом локальной обработки.

1.2 Клиенты и серверы

Автоматическое разделение клиентских и серверных узлов

Grid вводит понятие клиентских и серверных узлов. Серверные узлы участвуют в кэшировании, выполняют вычисления, обрабатывают поток данных и т.д., в то время как клиентские узлы обеспечивают удаленное подключение к серверам и выполняют запросы. Клиенты **Grid** позволяют использовать весь набор API-интерфейсов, предоставляемых **ADG**, включая кэширование, транзакции, вычисления, потоковую обработку, прочие сервисы, реализуемые со стороны клиентов и т.д.

По умолчанию все узлы **Grid** запускаются как узлы сервера, режим клиента должен быть явно включён при необходимости.

1.2.1 Настройка клиентов и серверов

Настроить узел как клиент или сервер можно через свойство *IgniteConfiguration.setClientMode(...)*:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <!-- Enable client mode. -->
    <property name="clientMode" value="true"/>
    ...
</bean>
```

- Java:

```
IgniteConfiguration cfg = new IgniteConfiguration();

// Enable client mode.
cfg.setClientMode(true);

// Start Ignite in client mode.
Ignite ignite = Ignition.start(cfg);
```

Кроме того, для удобства также можно включить или отключить режим клиента в классе *Ignition*, чтобы клиенты и серверы использовали одну и ту же конфигурацию:

- Java:

```
Ignition.setClientMode(true);

// Start Ignite in client mode.
Ignite ignite = Ignition.start();
```

1.2.2 Создание распределенных кэшей

Всякий раз при создании кэшей в **Grid** в **XML** либо при помощи метода *Ignite.createCache(...)* или *Ignite.getOrCreateCache(...)*, **ADG** автоматически развертывает распределенный кэш на всех узлах сервера.

Important: Как только распределенный кэш будет создан, он будет автоматически развернут на всех существующих и будущих узлах сервера

- Java:

```
// Enable client mode locally.
Ignition.setClientMode(true);
```

```
// Start Ignite in client mode.
Ignite ignite = Ignition.start();

CacheConfiguration cfg = new CacheConfiguration("myCache");

// Set required cache configuration properties.
...

// Create cache on all the existing and future server nodes.
// Note that since the local node is a client, it will not
// be caching any data.
IgniteCache<?, ?> cache = ignite.getOrCreateCache(cfg);
```

1.2.3 Вычисление на клиентах или серверах

По умолчанию *IgniteCompute* выполняет задания на всех узлах сервера. Но при этом можно выбрать выполнение заданий только на узлах сервера или только на клиентских узлах, создав соответствующую группу кластеров:

- Compute on Servers:

```
IgniteCompute compute = ignite.compute();

// Execute computation on the server nodes (default behavior).
compute.broadcast(() -> System.out.println("Hello Server"));
```

- Compute on Clients:

```
ClusterGroup clientGroup = ignite.cluster().forClients();

IgniteCompute clientCompute = ignite.compute(clientGroup);

// Execute computation on the client nodes.
clientCompute.broadcast(() -> System.out.println("Hello Client"));
```

1.2.4 Управление “медленными” клиентами (Slow Clients)

Во многих случаях развертывания клиентские узлы запускаются вне основного кластера на более медленных машинах с худшей сетью. При этом возможно, что серверы будут генерировать нагрузку (например, уведомления о непрерывных запросах), которую клиенты не смогут обрабатывать, что в свою очередь приводит к увеличению очереди исходящих сообщений на серверах. Это может в конечном счете вызвать либо ситуацию с недостаточным объемом памяти на сервере, либо заблокировать весь кластер (если включен контроль обратного давления).

Для управления подобными ситуациями можно настроить максимальное количество разрешенных исходящих сообщений для клиентских узлов. Тогда в случае если размер исходящей очереди превышает указанное значение, клиентский узел отключается от кластера, предотвращая глобальное замедление.

В приведенных ниже примерах показано, как настроить ограничение очереди Slow Clients в коде и XML-конфигурации:

- Java:

```
IgniteConfiguration cfg = new IgniteConfiguration();

// Configure Ignite here.

TcpCommunicationSpi commSpi = new TcpCommunicationSpi();
commSpi.setSlowClientQueueLimit(1000);

cfg.setCommunicationSpi(commSpi);
```

- XML:

```
<bean id="grid.cfg" class="org.apache.ignite.configuration.IgniteConfiguration">
  <!-- Configure Ignite here. -->

  <property name="communicationSpi">
    <bean class="org.apache.ignite.spi.communication.tcp.TcpCommunicationSpi">
      <property name="slowClientQueueLimit" value="1000"/>
    </bean>
  </property>
</bean>
```

1.2.5 Повторное подключение клиента

Клиентские узлы могут быть отключены от кластера в нескольких случаях:

- Когда клиентский узел не может восстановить соединение с узлом сервера из-за сетевых проблем;
- Соединение с узлом сервера нарушено в течение некоторого времени; клиентский узел может восстановить соединение с сервером, но сервер уже удалил клиентский узел, так как не получал сигнал;
- “Медленные” клиенты могут быть отброшены узлами сервера.

Когда клиент устанавливает отключение от кластера, он присваивает узлу новый идентификатор “id” и пытается подключиться к кластеру. Важно знать, что это имеет побочный эффект - свойство “id” локального *ClusterNode* изменится в случае повторного подключения клиента. Это означает, что может быть затронута любая логика приложения, основанная на значении “id”.

Пока клиент находится в отключенном состоянии, и выполняется попытка повторного подключения, API-интерфейс **Grid** генерирует специальное исключение - *IgniteClientDisconnectedException*. Это исключение обеспечивает *future*, которое будет завершено при повторном подключении клиента к кластеру (API-интерфейс *IgniteCache* выдает *CacheException*, у которого в этом случае *IgniteClientDisconnectedException*). Это *future* можно также получить с помощью метода *IgniteCluster.clientReconnectFuture()*.

Кроме того, для повторного подключения клиента существуют специальные события (эти события являются локальными, то есть они запускаются только на клиентском узле):

- *EventType.EVT_CLIENT_NODE_DISCONNECTED*
- *EventType.EVT_CLIENT_NODE_RECONNECTED*

В следующем примере показано, как использовать *IgniteClientDisconnectedException*:

- Compute:

```
IgniteCompute compute = ignite.compute();

while (true) {
    try {
        compute.run(job);
```

```
    }
    catch (IgniteClientDisconnectedException e) {
        e.reconnectFuture().get(); // Wait for reconnection.

        // Can proceed and use the same IgniteCompute instance.
    }
}
```

- Cache:

```
IgniteCache cache = ignite.getOrCreateCache(new CacheConfiguration<>());

while (true) {
  try {
    cache.put(key, val);
  }
  catch (CacheException e) {
    if (e.getCause() instanceof IgniteClientDisconnectedException) {
      IgniteClientDisconnectedException cause =
        (IgniteClientDisconnectedException)e.getCause();

      cause.reconnectFuture().get(); // Wait for reconnection.

      // Can proceed and use the same IgniteCache instance.
    }
  }
}
```

Автоматическое повторное подключение клиента можно отключить с помощью свойства *clientReconnectDisabled* на *TcpDiscoverySpi*. Если переподключение отключено, клиентский узел останавливается.

- Java:

```
IgniteConfiguration cfg = new IgniteConfiguration();

// Configure Ignite here.

TcpDiscoverySpi discoverySpi = new TcpDiscoverySpi();

discoverySpi.setClientReconnectDisabled(true);

cfg.setDiscoverySpi(discoverySpi);
```

1.2.6 Принудительный режим сервера на клиентских узлах

Для запуска клиентских узлов требуются активные серверные узлы.

Однако, чтобы запустить клиентский узел без запущенного узла сервера, можно принудительно выполнить режим сервера на клиентских узлах следующим образом:

- Java:

```
IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setClientMode(true);

// Configure Ignite here.
```

```
TcpDiscoverySpi discoverySpi = new TcpDiscoverySpi();

discoverySpi.setForceServerMode(true);

cfg.setDiscoverySpi(discoverySpi);
```

В этом случае запуск будет происходить так, как если бы все узлы были активными узлами сервера. При этом все адреса, используемые **SPI** запуск на всех узлах, должны быть взаимно достижимы.

Глава 2

Durable Memory

- Durable Memory
- *Архитектура памяти*
- *Конфигурация памяти*

2.1 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.

2.1.1 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.

2.1.2 Функции персистентности

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

- *Grid Native Persistence*
- *Сторонняя персистентность*
- *Пространство подкачки памяти*

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.

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.

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.

2.2 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*

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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.

2.3 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);
```

Более подробные настройки долговременной памяти приведены в следующих разделах:

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

2.3.1 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.

Таблица2.1.: 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);
```

2.3.2 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.

2.3.3 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

Глава 3

Хранилище “ключ-значение”

Репликация и секционирование данных в памяти кластера

- *Data Grid*
- *Запросы к данным в кэше*
- *Режимы кэша*
- *Основные и резервные копии*

3.1 Data Grid

Платформа **Data Grid** построена с нуля с концепцией горизонтального масштабирования и возможностью добавления узлов по требованию в режиме реального времени; платформа разработана для линейного масштабирования до сотен узлов с сильной семантикой для маршрутизации данных с целью сокращения избыточного шума данных.

Data Grid – это *in-memory distributed key-value store* (распределенное хранилище ключ-значение в памяти), которое можно рассматривать как распределенную секционированную хэш-карту с каждым узлом кластера, владеющим частью общих данных. Таким образом, чем больше добавляется узлов кластера, тем больше данных можно кэшировать.

В отличие от других хранилищ “ключ-значение”, **Grid** определяет местоположение данных с помощью алгоритма хэширования. Каждый клиент может определить, к какому узлу принадлежит ключ, подключив его к функции хэширования, без необходимости каких-либо mapping servers или name nodes.

Data Grid поддерживает локальные, реплицированные и секционированные наборы данных и позволяет свободно выполнять перекрестные запросы между этими наборами данных, используя стандартный синтаксис **SQL**.

Grid поддерживает стандартный **SQL** для запросов данных, включая поддержку распределенных SQL-соединений.

Data Grid – молниеносная платформа, являющаяся одной из самых быстрых по реализации транзакции данных в кластере сегодня.

Data Grid реализует спецификацию **JCache** (**JSR 107**).

Important: Пока кластер работает, Grid гарантирует, что данные между различными узлами кластера всегда

будут оставаться неизменными независимо от сбоев или изменений топологии

3.2 Запросы к данным в кэше

Grid поддерживает очень элегантный **API** запросов с поддержкой запросов сканирования на основе предикатов, запросов **SQL** (совместимых с **ANSI-99**) и текстовых запросов. Для SQL-запросов **Grid** поддерживает индексирование в памяти, поэтому все поисковые запросы чрезвычайно быстры.

Grid также обеспечивает поддержку пользовательской индексации с помощью классов *IndexingSpi* и *SpiQuery*.

IgniteCache имеет несколько методов запросов, все из которых получают некоторый подкласс класса *Query* и возвращают *QueryCursor*.

Абстрактный класс *Query* представляет собой абстрактный разбитый на страницы запрос для выполнения в распределенном кэше. Размер страницы можно задать с помощью метода *Query.setPageSize(. . .)*, значение по умолчанию которого – *1024*.

QueryCursor представляет собой набор результатов запроса и позволяет проводить прозрачную постраничную итерацию. Всякий раз, когда пользователь начинает итерацию последней страницы, в фоновом режиме автоматически запрашивается следующая страница. В случаях, когда разбиение на страницы не требуется, можно использовать метод *QueryCursor.getAll()*, который извлекает весь результат запроса и хранит его в коллекции.

Important: Курсоры закрываются автоматически при вызове метода *QueryCursor.getAll()*. При итерации по курсору в цикле или при прямом получении *Iterator*, необходимо явно закрыть курсор или использовать синтаксис *AutoCloseable*

Запросы сканирования позволяют запрашивать кэш в распределенной форме на основе определенного пользователем предиката:

- scan:

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Find only persons earning more than 1,000.
try (QueryCursor cursor = cache.query(new ScanQuery((k, p) -> p.getSalary() > 1000)) {
  for (Person p : cursor)
    System.out.println(p.toString());
}
```

- java7 scan:

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Find only persons earning more than 1,000.
IgniteBiPredicate<Long, Person> filter = new IgniteBiPredicate<>() {
  @Override public boolean apply(Long key, Perons p) {
     return p.getSalary() > 1000;
     }
};

try (QueryCursor cursor = cache.query(new ScanQuery(filter)) {
  for (Person p : cursor)
```

```
    System.out.println(p.toString());
}
```

Запросы сканирования также поддерживают дополнительное закрытие трансформатора, которое позволяет преобразовать запись на узле сервера перед ее отправкой клиенту. Это полезно, например, когда требуется извлечь только несколько полей из большого объекта и минимизировать сетевой трафик. Пример ниже показывает, как извлекать только ключи и не отправлять объекты.

- scan with transformer:

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Get only keys for persons earning more than 1,000.
List<Long> keys = cache.query(new ScanQuery<Long, Person>(
    (k, p) -> p.getSalary() > 1000), // Remote filter.
    Cache.Entry::getKey              // Transformer.
).getAll();
```

- scan with transformer (Java 7):

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Get only keys for persons earning more than 1,000.
List<Long> keys = cache.query(new ScanQuery<>(
    // Remote filter.
    new IgniteBiPredicate<Long, Person>() {
        @Override public boolean apply(Long k, Person p) {
            return p.getSalary() > 1000;
        }
    }),
    // Transformer.
    new IgniteClosure<Cache.Entry<Long, Person>, Long>() {
        @Override public Long apply(Cache.Entry<Long, Person> e) {
            return e.getKey();
        }
    }
).getAll();
```

SQL-запросы в **Grid** рассматриваются в разделе документации SQL.

Grid также поддерживает текстовые запросы, основанные на **Lucene** индексировании.

- text query:

```
IgniteCache<Long, Person> cache = ignite.cache("mycache");

// Query for all people with "Master Degree" in their resumes.
TextQuery txt = new TextQuery(Person.class, "Master Degree");

try (QueryCursor<Entry<Long, Person>> masters = cache.query(txt)) {
  for (Entry<Long, Person> e : cursor)
    System.out.println(e.getValue().toString());
}
```

3.2.1 Настройка запросов по аннотациям

Индексы могут быть сконфигурированы из кода с помощью *@QuerySqlField* аннотаций. Чтобы сообщить **Grid**, какие типы следует индексировать, пары ключ-значение можно передать методу *CacheConfiguration.setIndexedTypes(MyKey.class, MyValue.class)*. Данный метод принимает только пары типов – один для класса ключей и другой для класса значений.

- Java:

```
public class Person implements Serializable {
  /** Person ID (indexed). */
  @QuerySqlField(index = true)
  private long id;

  /** Organization ID (indexed). */
  @QuerySqlField(index = true)
  private long orgId;

  /** First name (not-indexed). */
  @QuerySqlField
  private String firstName;

  /** Last name (not indexed). */
  @QuerySqlField
  private String lastName;

  /** Resume text (create LUCENE-based TEXT index for this field). */
  @QueryTextField
  private String resume;

  /** Salary (indexed). */
  @QuerySqlField(index = true)
  private double salary;

  ...
}
```

3.2.2 Настройка запросов с помощью QueryEntity

Индексы и поля также можно настроить с помощью *org.apache.ignite.cache.QueryEntity*, удобным для конфигурации **XML** с **Spring**. Это эквивалентно использованию аннотации *@QuerySqlField*, поскольку аннотации классов преобразуются в сущности внутреннего запроса.

- XML:

```
<bean class="org.apache.ignite.configuration.CacheConfiguration">
    <property name="name" value="mycache"/>
    <!-- Configure query entities -->
    <property name="queryEntities">
        <list>
            <bean class="org.apache.ignite.cache.QueryEntity">
                <property name="keyType" value="java.lang.Long"/>
                <property name="valueType" value="org.apache.ignite.examples.Person"/>

                <property name="fields">
                    <map>
                        <entry key="id" value="java.lang.Long"/>
```

```
                        <entry key="orgId" value="java.lang.Long"/>
                        <entry key="firstName" value="java.lang.String"/>
                        <entry key="lastName" value="java.lang.String"/>
                        <entry key="resume" value="java.lang.String"/>
                        <entry key="salary" value="java.lang.Double"/>
                    </map>
                </property>

                <property name="indexes">
                    <list>
                        <bean class="org.apache.ignite.cache.QueryIndex">
                            <constructor-arg value="id"/>
                        </bean>
                        <bean class="org.apache.ignite.cache.QueryIndex">
                            <constructor-arg value="orgId"/>
                        </bean>
                        <bean class="org.apache.ignite.cache.QueryIndex">
                            <constructor-arg value="salary"/>
                        </bean>
                    </list>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

- Java:

```
CacheConfiguration<Long, Person> cacheCfg = new CacheConfiguration<>();
...
cacheCfg.setName("mycache");

// Setting up query entity.
QueryEntity queryEntity = new QueryEntity();

queryEntity.setKeyType(Long.class.getName());
queryEntity.setValueType(Person.class.getName());

// Listing query fields.
LinkedHashMap<String, String> fields = new LinkedHashMap();

fields.put("id", Long.class.getName());
fields.put("orgId", Long.class.getName());
fields.put("firstName", String.class.getName());
fields.put("lastName", String.class.getName());
fields.put("resume", String.class.getName());
fields.put("salary", Double.class.getName());

queryEntity.setFields(fields);

// Listing indexes.
Collection<QueryIndex> indexes = new ArrayList<>(3);

indexes.add(new QueryIndex("id"));
indexes.add(new QueryIndex("orgId"));
indexes.add(new QueryIndex("salary"));

queryEntity.setIndexes(indexes);
```

```
...
cacheCfg.setQueryEntities(Arrays.asList(queryEntity));
...
```

3.3 Режимы кэша

ADG предоставляет три разных режима работы кэша: *PARTITIONED*, *REPLICATED* и *LOCAL*. Режим настраивается для каждого кэша и определяется в перечислении *CacheMode*.

3.3.1 Режим Partitioned

Режим *PARTITIONED* – это наиболее масштабируемый режим распределенного кэша. В этом режиме общий набор данных разделяется поровну на секции, и все секции распределяются поровну между участвующими узлами, создавая, по сути, одно огромное распределенное хранилище данных. Такой подход позволяет хранить как можно больше данных в общей памяти (RAM и диск), доступной для всех узлов. Получается, чем больше узлов, тем больше данных можно хранить.

В отличие от режима *REPLICATED*, в котором обновления являются дорогостоящими, поскольку каждому узлу в кластере требуется обновление, в режиме *PARTITIONED* обновления дешевле, потому что для каждого ключа необходимо обновлять только один главный узел (и опционально один или несколько резервных узлов). Тем не менее, чтение становится несколько дороже, так как только некоторые узлы кэшируют данные.

Во избежание дополнительного перемещения данных важно всегда обращаться к данным именно на том узле, который их кэширует. Этот подход называется *affinity colocation* и настоятельно рекомендуется при работе с сегментированными кэшами.

Important: Сегментированные кэши идеально подходят для работы с большими наборами данных и при их частых обновлениях

Пример настройки режима кэширования *PARTITIONED* приведен в разделе *Конфигурация режимов кэширования*.

3.3.2 Режим Replicated

В режиме *REPLICATED* все данные реплицируются на каждый узел кластера. Этот режим кэширования обеспечивает максимальную доступность данных, поскольку он доступен на каждом узле. Однако, в этом режиме каждое обновление данных должно распространяться на все остальные узлы, что может повлиять на производительность и масштабируемость.

В **Grid** реплицированные кэши реализуются аналогично сегментированным, где каждый ключ имеет основную копию, а также копируется на всех других узлах кластера.

Поскольку одни и те же данные хранятся на всех узлах кластера, размер реплицированного кэша ограничен объемом памяти (RAM и диск), доступным на узле. Этот режим идеально подходит для сценариев, в которых операции чтения кэша выполняются намного чаще, чем операции записи, а наборы данных при этом небольшие. Если система выполняет поиск в кэше более *80%* времени, следует использовать режим кэширования *REPLICATED*.

Important: Реплицированные кэши следует использовать при малых наборах данных и при их нечастых обновлениях

3.3.3 Режим Local

Режим *LOCAL* – самый легкий режим работы кэша, поскольку данные не распределяются по другим узлам кэша. Он идеально подходит для сценариев, где данные либо доступны только для чтения, либо могут периодически обновляться с определенной частотой истечения срока действия. Режим также отлично работает с сквозным чтением, когда данные загружаются из постоянного хранилища. Локальные кэши имеют все функции распределенного кэша, такие как автоматическое вытеснение данных, истечение срока действия, смена дисков, запрос данных и транзакции.

3.3.4 Конфигурация режимов кэширования

Режимы кэша настраиваются для каждого кэша путем установки в параметр *CacheConfiguration* свойства *cacheMode* следующим образом:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
    ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
            <!-- Set a cache name. -->
            <property name="name" value="cacheName"/>
            <!-- Set cache mode. -->
            <property name="cacheMode" value="PARTITIONED"/>
            <!-- Other cache configurations. -->
            ...
        </bean>
    </property>
</bean>
```

- Java:

```
CacheConfiguration cacheCfg = new CacheConfiguration("myCache");

cacheCfg.setCacheMode(CacheMode.PARTITIONED);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

3.4 Основные и резервные копии

В режиме *PARTITIONED* узлы, которым назначены ключи, называются основными узлами для этих ключей. Кроме того, можно дополнительно настроить любое количество резервных узлов для кэшированных данных. Если количество резервных копий больше *0*, **Grid** автоматически назначает узлы резервного

копирования для каждого отдельного ключа. Например, если количество резервных копий равно *1*, то каждый кэшированный ключ имеет *2* копии: **1 основную и *1* резервную.

Important: По умолчанию резервное копирование отключено с целью повышения производительности

Резервные копии настраиваются установкой свойства *backups()* в параметре *CacheConfiguration*, например:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
     ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
             <!-- Set a cache name. -->
             <property name="name" value="cacheName"/>

             <!-- Set cache mode. -->
                            <property name="cacheMode" value="PARTITIONED"/>

                <!-- Number of backup nodes. -->
                               <property name="backups" value="1"/>
                            ...
        </bean>
    </property>
</bean>
```

- Java:

```
CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

cacheCfg.setCacheMode(CacheMode.PARTITIONED);

cacheCfg.setBackups(1);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

3.4.1 Синхронное и асинхронное резервное копирование

CacheWriteSynchronizationMode можно использовать для настройки синхронного или асинхронного обновления основных и резервных копий. Режим синхронизации записи сообщает **ADG**, должен ли клиентский узел ожидать ответов от удаленных узлов до завершения записи или коммита.

Синхронизация записи может быть установлена в одном из следующих трех режимов:

- *FULL_SYNC* – клиентский узел ожидает завершения записи или коммита на всех участвующих удаленных узлах (основной и резервный);
- *FULL_ASYNC* – клиентский узел не ожидает ответов от участвующих узлов, и удаленные узлы могут обновить свое состояние после завершения записи кэша любым из методов или после завершения метода *Transaction.commit()*;

- *PRIMARY_SYNC* – режим по умолчанию. Клиентский узел ожидает завершения записи или комиита на основном узле, но не ждет обновления резервных копий.

Important: Независимо от режима синхронизации записи данные кэша всегда остаются полностью согласованными на всех участвующих узлах при использовании транзакций

Режим синхронизации записи настраивается установкой свойства *writeSynchronizationMode* в параметре *CacheConfiguration*, например:

- XML:

```
<bean class="org.apache.ignite.configuration.IgniteConfiguration">
        ...
    <property name="cacheConfiguration">
        <bean class="org.apache.ignite.configuration.CacheConfiguration">
             <!-- Set a cache name. -->
             <property name="name" value="cacheName"/>

                <!-- Set write synchronization mode. -->
                             <property name="writeSynchronizationMode" value="FULL_SYNC"/>
                             ...
        </bean>
    </property>
</bean>
```

- Java:

```
CacheConfiguration cacheCfg = new CacheConfiguration();

cacheCfg.setName("cacheName");

cacheCfg.setWriteSynchronizationMode(CacheWriteSynchronizationMode.FULL_SYNC);

IgniteConfiguration cfg = new IgniteConfiguration();

cfg.setCacheConfiguration(cacheCfg);

// Start Ignite node.
Ignition.start(cfg);
```

Глава 4

Загрузка и стриминг данных

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.

4.1 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.

4.1.1 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

4.1.2 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);
    }
  }
  ...
}
```

4.1.3 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

4.2 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**.

4.2.1 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*

4.2.2 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(. . .)*

4.2.3 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);
```

4.2.4 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);
}
```