Arenadata™ Grid

Версия - v2.7

Основные концепции Arenadata Grid

2020 Arenadata™

Оглавление

1 Факты об Arenadata Grid **3**

2 Клиенты и серверы **5**

2.1 Настройка клиентов и серверов 5
2.2 Создание распределенных кэшей 6
2.3 Вычисление на клиентах или серверах 6
2.4 Управление “медленными” клиентами (Slow Clients) 7
2.5 Повторное подключение клиента 7
2.6 Принудительный режим сервера на клиентских узлах 9

В документации приведены основные факты об Arenadata Grid и описана работа с клиентами и серверами.

Инструкция может быть полезна администраторам, программистам, разработчикам и сотрудникам подразделений информационных технологий, осуществляющих внедрение системы. Раздел предлагается к прочтению перед переходом к непосредственной установке сервера.

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

Глава 1

Факты об 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.

- 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?

Да. Начиная с версии 2.7 Grid поддерживает транзакции, реализуя MVCC.

Транзакции **ACID** также поддерживаются при работе непосредственно с KV-кешем. **Grid** поддерживает транзакции между партициями, что означает, что транзакции могут охватывать ключи, находящиеся в разных партициях на разных серверах.

- 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.

Глава 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 даннных 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.

- *Настройка клиентов и серверов*
- *Создание распределенных кэшей*
- *Вычисление на клиентах или серверах*
- **‘Управление “медленными” клиентами (Slow Clients)‘_**
- *Повторное подключение клиента*
- *Принудительный режим сервера на клиентских узлах*

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();
```

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);
```

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"));
```

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>
```

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);
```

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.