Arenadata™ Grid

Версия - v2.7

Хранилище "ключ-значение"в Arenadata Grid

2020 Arenadata™

Оглавление

1 Data Grid **3**

2 Запросы к данным в кэше **4**
2.1 Настройка запросов по аннотациям 6
2.2 Настройка запросов с помощью QueryEntity 6

3 Режимы кэша **9**
3.1 Режим Partitioned 9
3.2 Режим Replicated 9
3.3 Режим Local 10
3.4 Конфигурация режимов кэширования 10

4 Основные и резервные копии **11**
4.1 Синхронное и асинхронное резервное копирование 12

Репликация и секционирование данных в памяти кластера

В документации приведено описание используемого в сервере Arenadata Grid хранилища, возможные запросы к данным в кэше и его режимы и представлена информация об основном и резервном копировании.

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

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

Глава 1

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

Глава 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());
}
```

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;

  ...
}
```

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));
...
```

Глава 3

Режимы кэша

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

3.1 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*.

3.2 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 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.

3.4 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);
```

Глава 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);
```

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);
```