TCStore API Developer Guide

Version 10.7

October 2020

TERRACOTTA

This document applies to Terracotta 10.7 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: TC-STO-DG-107-20201015

Table of Contents

About This Documentation

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.aspx and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at http://techcommunity.softwareag.com. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

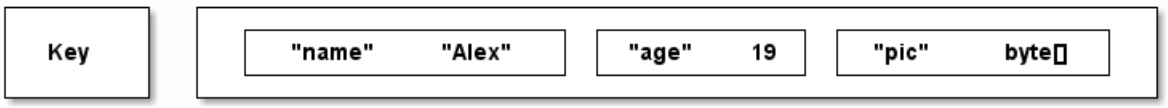
1 Reference

Concepts

Data Model

Data is organized by TCStore into collections called datasets. Each `Dataset` is comprised of zero or more records. Each`Record` has a key, unique within the dataset, and zero or more cells. Each `Cell` has a name, unique within the record; a declared type; and a non-null value. While records within a dataset must have a uniform key type, records are not required to have uniform content - each record may be comprised of cells having different names and/or different types. Each record and each cell is *self-describing* and is understood by the storage engine.

TCStore Data Storage Model - typed data.



Using popular/industry definitions, TCStore is an "Aggregate oriented, Key-Value NoSQL store". As noted above, the individual values stored within TCStore contain cells with type information enabling the store to make use of the data it holds. However, like other NoSQL stores, TCStore is schema-less in its core design, allowing individual entries to contain identical sets of cells, a subset of common cells, or a completely different sets of cells.

As such, and like other NoSQL stores, TCStore is not intended for usage patterns that are traditional to tabular data or RDBMSs. Data contained within TCStore are not and cannot be directly relational, and care should be taken to use modeling techniques (such as de-normalization of data) other than those commonly used with RDBMSs.

Type system

Fundamental to TCStore is the *type system* used in the data model.

The supported data types are:

- `BOOL`: A boolean value (either true or false), mapping to `java.lang.Boolean`; the `BOOL` type is associated with cells of type `Cell<Boolean>` and cell definitions of type `BoolCellDefinition` and `CellDefinition<Boolean>`
- `BYTES`: An array of bytes, signed 8-bit each, mapping to `byte[]`; the `BYTES` type is associated with cells of type `Cell<byte[]>` and cell definitions of type `BytesCellDefinition` and `CellDefinition<byte[]>`
- `CHAR`: A single UTF-16 character, 16-bit unsigned, mapping to `java.lang.Character`; the `CHAR` type is associated with cells of type `Cell<Character>` and cell definitions of type `CharCellDefinition` and `CellDefinition<Character>`
- `DOUBLE`: A 64-bit floating point value, mapping to `java.lang.Double`; the `DOUBLE` type is associated with cells of type `Cell<Double>` and cell definitions of type `DoubleCellDefinition` and `CellDefinition<Double>`

- `INT`: A signed 32-bit integer value, mapping to `java.lang.Integer` ; the `INT` type is associated with cells of type `Cell<Integer>` and cell definitions of type `IntCellDefinition` and `CellDefinition<Integer>`
- `LIST`: An ordered, randomly accessible list of `TypedValue` objects; that is, a list of value-type pairs. Each entry can be any of the other types including `LIST`. The `LIST` type is associated with cells of type `Cell<StoreList>` and cell definitions of type `ListCellDefinition` and `CellDefinition<StoreList>`. `LIST` is one of the two Complex Data types.
- `LONG`: A signed 64-bit integer value, mapping to `java.lang.Long`; the `LONG` type is associated with cells of type `Cell<Long>` and cell definitions of type `LongCellDefinition` and `CellDefinition<Long>`
- `MAP`: A String-keyed associative map of `TypedValue` objects; that is, a map of Strings to value-type pairs. Each entry value can be any of the other types including `MAP`. The `MAP` type is associated with cells of type `Cell<StoreMap>` and cell definitions of type `MapCellDefinition` and `CellDefinition<StoreMap>`. `MAP` is one of the two Complex Data types.
- `STRING`: A variable length sequence of `CHAR`, mapping to `java.lang.String`; the `STRING` type is associated with cells of type `Cell<String>` and cell definitions of type `StringCellDefinition` and `CellDefinition<String>`

The key of a `Record` may be an instance of any of the above types except `BYTES`, `LIST`, or `MAP`. The value of a `Cell` may be an instance of any one of the above types.

Datasets

A Dataset is a (possibly distributed), collection of `Record` instances. Each `Record` instance is uniquely identified by a key within the `Dataset`. The key type is declared when the `Dataset` is created. Aside from the `Record` key type, a `Dataset` has no predefined schema.

Records

A `Record` is a key plus an unordered set of "name to (typed) value" pairs representing a natural aggregate of your domain model. Each `Record` within a `Dataset` can hold completely different sets of name/value pairs, as there is no schema to obey. `Record` instances held within a given `Dataset` are immutable. Changing one or multiple values on a `Record` creates a new instance of that `Record` which replaces the old instance.

`Record` represents the only atomically alterable type in TCStore. You can mutate as many `Cell` instances of a given `Record` instance as you wish as an atomic action, but atomic actions cannot encompass more than one record.

Cell definitions and values

A `Record` contains zero or more `Cell` instances, each derived from a `CellDefinition`. A `CellDefinition` is a "type/name" pair (e.g. `String firstName`). From a `CellDefinition` you can create a `Cell` (e.g. `firstName = "Alex"`, where "Alex" is of type `String`) to store in a `Record`. The name of the `Cell` is the name from the `CellDefinition` used to create the cell; the value of the `Cell` is of the type specified in the `CellDefinition` used to create the cell.

`Cell` instances cannot contain null values but, since every `Record` in the dataset need not have uniform content, a `Cell` instance may be omitted from a `Record` for which that cell has no value. The API will then let you test a `Record` for the absence of that cell.

Note:
The `Cell` instances within a `Record` are unordered.

Working with Complex Data Types

Two of the TCStore types are *Complex Data Types* (CDTs): `LIST` and `MAP`. These are represented programmatically as instances of `StoreList` and `StoreMap`. CDTs share some common attributes:

- the values in CDTs are comprised of `TypedValue` instances, consisting of TCStore `Type` and a value of the corresponding type.
- CDTs can contain other CDT values, as well as the more typical scalar types.
- they can be immutable or mutable, and a new deep-copy mutable instance can be created from any instance via the `mutableCopy()` method.
- all support `find(spec) Selection` querying for efficient selecting of arbitrary data inside CDTs by type, position, and key, in various combinations.
- they can be exported to a text representation via the `toString(boolean skipWhitespace)` method
- the text representation can be parsed into complex data objects via the `ComplexDataParser` class, as well as various helper methods on `StoreList`, `StoreMap`, and `ComplexRecord`.

The `StoreMap`/`StoreList` hierarchy can be thought of as a JSON-like data structure with more specific typing (JSON Number becomes `Integer`, `Long`, `Double`,) and extra types (`Character`, `byte[]`), and no null values.

Important Classes

TypedValue

As stated, CDTs are comprised of `TypedValue` instances. `Cell` is a subclass of `TypedValue` as well, and bare `TypedValue` instances can be constructed by various helper methods on `TypedValue`.

StoreMap

The `StoreMap` interface defines a Java Map with String keys and `TypedValue` values. `StoreMap` objects come in two flavors, immutable and mutable. Typically, maps retrieved from map cells in the store will be immutable.

StoreList

The `StoreList` interface defines a Java List with `TypedValue` values. `StoreList` objects come in two flavors, immutable and mutable. Typically, lists retrieved from list cells in the store will be immutable.

ComplexRecord

`ComplexRecord` instances are synthetic representations of `Record` instances. Logically, any `Record` can be thought of as a `ComplexMap` plus the key for the `Record` (in fact, `ComplexRecord` implements `ComplexMap` just like `StoreMap` does). Any `Record` can be accessed as a `ComplexRecord` via the `Record.asComplex()` method. `ComplexRecord` objects retain the key typing of their source `Record`.

Finally, any `ComplexRecord` can be turned into a `CellSet`, suitable for adding to a `Dataset` (via the same method as `StoreMap` uses), or can be turned back into an immutable `Record` via the `asRecord()` helper method.

Construction and Manipulation of CDTs

CDTs fetched from a `Dataset` will be immutable. There are a variety of ways to make mutable copies or to create your own CDTs. Once a `StoreList` or `StoreMap` is mutable, it can be manipulated as if was an everyday Java List or Map, respectively.

StoreMap

To create a mutable `StoreMap`, use the `StoreMap.newMap()` helper method. Additionally, `StoreMap` instances can be constructed by parsing the text representation using the `mapOf()` helper methods. A `StoreMap` can also be constructed from a `CellSet` (remember, `Record` instances are `CellSets`) using a `mapOf()` helper.

Mutable `StoreMap` instances have functional `with(xxx)` methods for fluent construction.

For example, this code:

```
StoreMap m = StoreMap.newMap()
      .with("name", "john")
      .with("initial", 'R')
      .with("age", 10)
      .with("alive", true)
      .with("marker", 100101010L)
      .with("temp", 98.6d)
      .with("img", new byte[] { 1, 2, 3, 4 });
```

produces this structure:

```
{
  "name" : string : "john",
  "initial" : char : 'R',
  "age" : int : 10,
  "alive" : bool : true,
  "marker" : long : 100101010,
  "temp" : double : 98.6d,
  "img" : bytes : AQIDBA==
}
```

StoreList

To create a mutable `StoreList`, use the `StoreList.newMap()` helper method. Additionally, `StoreList` instances can be constructed by parsing the text representation using the `mapOf()` helper methods.

Mutable `StoreList` instances have functional `with(xxx)` methods for fluent construction.

So this code:

```
 StoreList l = StoreList.newList()
     .with("john")
     .with('R')
     .with(10)
     .with(true)
     .with(100101010L)
     .with(98.6d)
     .with(new byte[] { 1, 2, 3, 4 })
```

produces this structure:

```
[
  string : "john",
  char : 'R',
  int : 10,
  bool : true,
  long : 100101010,
  double : 98.6d,
  bytes : AQIDBA==
]
```

ComplexRecord

Immutable `ComplexRecord` objects can be made mutable, just like other CDTs, via a `mutableCopy()` method. Changes to such a mutable copy are not, of course, reflected in the original source record.

Furthermore, mutable `ComplexRecord` objects can be constructed via the `ComplexRecord.mutableRecord(K key, StoreMap cells)` method. In this case, further modification is available through the same `with(xxx)` methods as `StoreMap` provides.

Filtering and Querying CDTs

Since CDTs can be arbitrarily complex, a method is needed to concisely identify subsets of their values for further evaluation. The `find()` method, coupled with the `Selection` class for holding filtered views of a CDT, provide a powerful mechanism for evaluating the contents of CDTs.

Selection Objects, a Simple Example

To provide concise selection, CDTs support a text based selection methodology, coupled with programmatic refinement of the results.

For example, assume a map structure like this:

```
{
   "name" : map : {
      "first" : string : "chris",
      "last" : string : "jones",
      "age" : int : 20,
      "degrees" : list : [ "bs", "ms", "phd" ]
   }
}
```

and this code fragment:

```
StoreMap m;
Selection sel1 = m.find("name.first");
Selection sel2 = m.find("name.[]");
```

```
Selection sel3 = m.find("name.[]").ints();
Selection sel4 = m.find("name.[]").ints().get(0);
Selection sel5 = m.find("name.degrees.[]").strings().last();
Selection sel6 = m.find("nam.degrees.[]")
```

`sel1` will contain a single `TypedValue`, a String value of "chris".

`sel2` will contain all 4 `TypedValues`, in the `name` keyed map.

`sel3` will contain a single `TypedValue`, an integer value of 20.

`sel4` will contain a single `TypedValue`, an integer value of 20.

`sel5` will contain a single `TypedValue`, a String value of "phd".

`sel6` will be empty.

A `Selection` object can be:

- restricted by any of the defined types in TCStore,
- can be accessed randomly by an integer index to return an Optional
- can be accessed randomly by a negative integer to return a value offset from the last entry, where `get(-1)` accessed the last entry, `get(-2)` accesses the second from last, etc.
- first and last entries can be accessed directly.

`Selection` objects are instances of Java `Collection<TypedValue>`, so they can be iterated over, have `size()` and `contains()` methods, etc. Refining a `Selection` object by any of the Comparable types in TCStore (`selection.ints()`, `selection.strings()`, etc), will return a `ComparableSelection` object, which then adds things like `min()`, `max()`, `greaterThan()`, etc. Refining to a `Number` type via `ints()`, `longs()`, or `doubles()` will return a specifically typed `Number` selection, that is both `Comparable` and adds the numeric method `sum()`.

`Selection` object manipulation for querying is supported by the filtering DSL, and it is anticipated that much of this will be made portable for server side execution in the future.

find() and the dot-specification

In the above example, the dot specification was used to select a subset of the CDT. This concise, textual specification allows for the selection of subsets of CDTs in simple, intuitive manner. Simply put, a dot specification consists of a sequence of key **names** or **array slices**, separated by a period. So,

```
foo.[10].baz
```

starts at the outermost map, selects the value that has the key 'foo', treats it as a list, then uses the 11th indexed element (list references begin with 0) from the list, then treats the list element as a map, and selects the value with the key 'baz'.

Note that the manner of referencing at each level denotes typing as well; if the value found at 'foo' is not a list, then the `Selection` returned will be empty. `find()` calls cannot fail, except for parsing errors, they can however return an empty `Selection`.

Dot notation is intended to be both simple and powerful.

- '.' by itself matches the current object
- '.foo' if the top-level object is a map, matches the value of the 'foo' key
- '.foo.bar' matches the top-level object as a map, then the value at key 'foo', then matches if that value is a map, then matches the value of its key 'bar'. etc
- keys for matching can also be expressed with more explicit syntax for handling special characters:
  - '.[foo bar].[baz]' to allow for spaces
  - '.["foo\nbar"].baz' to allow for special characters
- '.[1]' matches the list entry at index 1 (lists are zero based)
- '.[1:3]' matches the slice from 1 to 3 inclusive
- '.[-2]' selects the second to last entry
- finally, a stanza of '.[]' matches all entries at this level

`find()` and its DSL variants all return `Selection` objects, which themselves allow further refining of the search results.

An important consideration when working with `Selection` objects and the DSL is that once you get to a typed scalar value (e.g., `...find("name").strings().first()`) you can then use that value as you would any other scalar value, for comparisons and such.

Configuration and Lifecycle Operations

Clustered DatasetManager using the API

Full example

The following sample code configures a new clustered `Dataset` with a system configured default persistent storage engine:

```
try (DatasetManager datasetManager = DatasetManager.clustered(clusterUri) // 1
     .build()) {                                                          // 2
  DatasetConfiguration ordersConfig =
      datasetManager.datasetConfiguration()                               // 3
      .offheap(offHeapResourceId)                                         // 4
      .disk(diskResourceId)                                               // 5
      .build();                                                           // 6
  datasetManager.newDataset("orders", Type.LONG, ordersConfig);           // 7
  try (Dataset orders =
      datasetManager.getDataset("orders", Type.LONG)) {                   // 8
    // Use the Dataset
  }
}
```

The following sample code configures a new clustered `Dataset` with an explicitly specified persistent storage engine:

```
try (DatasetManager datasetManager = DatasetManager.clustered(clusterUri) // 1
   .build()) {                                                             // 2
 DatasetConfiguration employeesConfig =
     datasetManager.datasetConfiguration()                                 // 3
     .offheap(offHeapResourceId)                                           // 4
     .disk(diskResourceId, PersistentStorageEngine.FRS)                    // 5
     .build();                                                             // 6
 datasetManager.newDataset("employees", Type.LONG, employeesConfig);       // 7
 try (Dataset<?> orders =
     datasetManager.getDataset("employees", Type.LONG)) {                  // 8
   // Use the Dataset
 }
}
```

1	The static method `DatasetManager.clustered` starts the process of configuring a clustered `DatasetManager`. It returns a `DatasetManagerBuilder` which allows configuration of the cluster client.
2	The `DatasetManager` is created, represents a connection to the cluster. `DatasetManager` is `AutoCloseable` so try-with-resources should be used.
3	A `DatasetConfiguration` is required to create a new `Dataset`. A `DatasetConfigurationBuilder` that can be used to construct a `DatasetConfiguration` is acquired using the method `datasetConfiguration` on the `DatasetManager`. Note that a `DatasetConfiguration` should be used with the `DatasetManager` that was used to create it.
4	A server side offheap resource is specified for data to be held in. Note that the name supplied must match the name of an offheap resource configured on the server.
5	A server side disk resource is specified for data to be held in. Note that the name supplied must match the name of a disk resource configured on the server. As illustrated in the two examples above, an optional persistent storage engine parameter can be specified along with the disk resource, denoting the underlying persistent storage engine technology that needs to used for this dataset. See the section "Note on the supported persistent storage engine technologies" on page 16 below for a discussion on currently supported persistent storage engine technologies.
6	The specification of the `DatasetConfiguration` is now completed and an instance is created.
7	A new `Dataset` called `orders` is created. It has a key of type `LONG`.
8	The previously created dataset is retrieved. `Dataset` is `AutoCloseable` so try-with-resources should be used.

URI to connect to server

The cluster URI takes the form of:

```
terracotta://<server1>:<port>,<server2>:<port>
```

for example:

```
terracotta://tcstore1:9510,tcstore2:9510
```

where `tcstore1` and `tcstore2` are the names of the servers that form the cluster.

Configuring a Dataset

When a `Dataset` is created, the name of the dataset and the type of the key must be specified. These are the first two parameters to `createDataset` and the same values should be used to later access the same `Dataset` via `getDataset`.

The third parameter is a `DatasetConfiguration` which specifies how storage for the `Dataset` should be managed on the server.

When the server is configured, any offheap memory resources or filesystem directories in which data can be written are given names. Any string passed to `offheap` or `disk` should match the name of a resource configured on the server. This resource will then be used for storage for the `Dataset`.

A `Dataset` must have an offheap resource configured for it. If the disk resource is specified then the records of the `Dataset` will be recorded on disk. If the disk resource is specified, the persistent storage engine technology used to persist on disk can also be optionally specified. If no persistent storage engine is specified, the default persistent storage engine will be used. If no disk resource is specified, then data is held just in the memory of the servers of the cluster.

Note on the supported persistent storage engine technologies

The currently supported persistent storage engine are as follows:

- PersistentStorageEngine.FRS
- PersistentStorageEngine.HYBRID

The persistent storage engines vary in the rules on how the persistent store is used. However all persistent storage engines provides strong guarantees on data being non-volatile and durable across server crashes and restarts.

1. FRS
   - The Heap (or offheap) is pushed to the FRS instance on disk, but processing is served from memory.
   - Primary, secondary, and heap and offheap memory structures are rebuilt from FRS on restart.
   - Primary and secondary indexes are rebuilt from scratch in memory on restart.
2. HYBRID also uses FRS technology underneath with the following caveats.
   - Here the Heap is pushed to FRS.
   - The in-memory heap is merely a mapping to locate a value given a key from the disk.
   - Record lookups are served by asking the storage engine.

- Primary, secondary structures reside in-memory

In the future, new persistent storage engine technologies could be supported and allowed to be configured for a dataset.

Current Limitations when configuring persistent storage engines

There are some limitations on how these storage engines can be configured against a dataset. In the future one or more of these limitations may be lifted.

- A given disk resource can only hold a single storage engine. This means two datasets using the same disk resource must specify the same storage engine technology.
- The current default persistent storage engine, if none is specified when dataset is configured, is FRS. Again if a dataset uses the disk resource with the default storage engine, another dataset using the same dataset must use the same storage engine.

Note on the fluent API

TCStore uses a fluent API to allow configuration calls to be chained. Following this pattern, each call returns a builder so that further configuration can be made, however, TCStore returns a different instance each time. This allows a `DatasetManagerBuilder` to be used as a prototype for different configurations, but this means that code such as:

```
ClusteredDatasetManagerBuilder builder = DatasetManager.clustered(clusterUri);
builder.withConnectionTimeout(30, TimeUnit.SECONDS);
DatasetManager datasetManager = builder.build();
```

will create a clustered `DatasetManager` that has the default connection timeout because `build` is called on the wrong object.

Instead use the following form:

```
ClusteredDatasetManagerBuilder builder = DatasetManager.clustered(clusterUri);
ClusteredDatasetManagerBuilder configuredBuilder =
    builder.withConnectionTimeout(30, TimeUnit.SECONDS);
DatasetManager datasetManager = configuredBuilder.build();
```

or more fluently:

```
DatasetManager datasetManager = DatasetManager.clustered(clusterUri)
        .withConnectionTimeout(30, TimeUnit.SECONDS)
        .build();
```

Clustered DatasetManager using XML

Creating a Clustered DatasetManager

```
URL configUrl = getClass().getResource("clustered.xml"); // 1
DatasetManagerConfiguration datasetManagerConfiguration =
   XmlConfiguration.parseDatasetManagerConfig(configUrl);        // 2
try (DatasetManager datasetManager =
   DatasetManager.using(datasetManagerConfiguration,
     ConfigurationMode.CREATE)) {                                // 3
```

```
        try (Dataset<Long> dataset = datasetManager.getDataset("orders",
            Type.LONG)) {                                          // 4
    // use the dataset
  }
}
```

1	Gets the clustered `DatasetManager` configuration URL (see "Clustered DatasetManager XML configuration" on page 18).
2	Creates a `DatasetManagerConfiguration` using the configuration URL.
3	Creates a clustered `DatasetManager` using the given `DatasetManagerConfiguration` and `ConfigurationMode` (see "Configuration modes" on page 20).
4	Gets the `Dataset` with name `orders` and type `LONG`.

Getting DatasetManager configuration in XML form

```
try (DatasetManager datasetManager = DatasetManager.clustered(
    URI.create("terracotta://localhost:9510")).build()) {                 // 1
  DatasetManagerConfiguration datasetManagerConfiguration =
     datasetManager.getDatasetManagerConfiguration();                     // 2
  String xmlConfig = XmlConfiguration.toXml(datasetManagerConfiguration)) // 3
}
```

1	Creates a clustered `DatasetManager`.
2	Gets the `DatasetManagerConfiguration` of the `DatasetManager`.
3	Converts the `DatasetManagerConfiguration` to XML form.

Clustered DatasetManager XML configuration

```
<clustered xmlns=
   "http://www.terracottatech.com/v1/terracotta/store/clustered"> <!-- 1 -->
  <cluster-connection>
    <server host="localhost" port="9510"/>                         <!-- 2 -->
    <connection-timeout unit="MILLIS">10</connection-timeout>      <!-- 3 -->
    <reconnection-timeout unit="MILLIS">20</reconnection-timeout> <!-- 4 -->
    <security-root-directory>/path/to/security-root-directory
        </security-root-directory>                                 <!-- 5 -->
    <client-alias>client-alias</client-alias>                      <!-- 6 -->
    <client-tags>client-tags</client-tags>                         <!-- 7 -->
  </cluster-connection>
  <dataset name="orders" key-type="LONG">                          <!-- 8 -->
    <!-- dataset configuration -->
  </dataset>
  <!-- other datasets -->
</clustered>
```

1	Declares a clustered `DatasetManager` configuration.

2	Configures a Terracotta server in the cluster (`port` attribute is optional, default is 9410)
3	Configures the connection timeout for this connection (optional, default 20 milliseconds).
4	Configures the reconnection timeout for this connection (optional, default 0 milliseconds).
5	Configures the security root directory to make a secure connection to the Terracotta cluster (optional).
6	Configures an alias to identify this client (optional, default is a randomly generated string).
7	Configures tags for this client (optional, default is no tags).
8	Declares a `Dataset` with name `orders`, key type `LONG` and its configuration (see "Dataset XML configuration" on page 19).

Dataset XML configuration

```
<dataset name="orders" key-type="LONG"
   xmlns:tcs="http://www.terracottatech.com/v1/terracotta/store"> <!--1-->
  <tcs:offheap-resource>offheap</tcs:offheap-resource>            <!--2-->
  <tcs:disk-resource storage-type="FRS">disk</tcs:disk-resource>  <!--3-->
  <tcs:indexes>
    <tcs:index>                                                   <!--4-->
      <tcs:cell-definition name="cell" type="BOOL"/>
      <tcs:type>BTREE</tcs:type>
    </tcs:index>
  </tcs:indexes>
  <tcs:durability-eventual/>                                      <!--5-->
  <tcs:advanced>                                                  <!--6-->
    <tcs:concurrency-hint>16</tcs:concurrency-hint>
  </tcs:advanced>
</dataset>
```

1	Declares a `Dataset` configuration.
2	Configures an offheap resource for this dataset.
3	Configures a disk resource for this dataset (optional) with an optional `storage-type` attribute that specifies the persistent storage engine to be used. The available engines are described in the section *Storage Types* below.
4	Configures an index for this dataset with a cell definition and its type (optional).
5	Configures disk durability for this dataset (optional).
6	Advanced `Dataset` configuration such as `concurrency-hint` (optional).

Configuration modes

Defines whether datasets configured in `DatasetManagerConfiguration` should be created or validated. The three supported configuration modes are:

1. `CREATE`
   - creates all configured datasets.
   - If any of the datasets already exists then the creation for that dataset fails, and no attempt is made to create any subsequent datasets in the list.
   - If the creation of a dataset fails, an exception is thrown containing the list of datasets that got created before it failed.
   - For example - Suppose we are trying to create three datasets named dataset1, dataset2 and datase3, and dataset2 exists already with the same key type or a different key type. In this scenario dataset1 will get created, but since dataset2 already exists, an exception will be thrown containing the message "A dataset with the name dataset2 already exists, following datasets were created so far: dataset1". No attempt will be made to create dataset3, since the creation stops at the exception for dataset2.
   - If the datasets creation using CREATE mode fails for any reason it is recommended to retry creating the datasets using AUTO mode so that any exceptions related to an already existing dataset can be avoided.
2. `VALIDATE`
   - validates all configured datasets.
   - If any of the datasets does not exist, or if it exists but has a different key type, then the validation fails and an exception is thrown. The exception contains a message stating whether the validation failed because the dataset does not exist, or because the dataset exists but has a different key type.
   - If the validation fails for a dataset, no validation is performed for the subsequent datasets in the list.
3. `AUTO`
   - creates datasets if any of the configured datasets don't exist, otherwise it validates the existing datasets.
   - For example - Suppose we are trying to create datasets named dataset1, dataset2 and dataset3, and dataset2 already exists with a different key type. In this scenario dataset1 will get created. Since dataset2 already exists, it will be validated, and an exception will be thrown since the key type is different. As a result, dataset3 will not be created because the exception for dataset2 stops the execution.

Storage Types

One of the following storage types can be specified for a given dataset.

- FRS - Specifies FRS storage engine
- HYBRID - Specifies HYBRID storage engine

Operations

Clustered Reconnection

When a TCStore `Dataset` is *cluster-resident*, operations on that Dataset involve interactions with the cluster servers holding that `Dataset`. These interactions occur over TCP connections to the servers in the cluster. Stability of these connections and the servers supporting them necessarily affects operations against the `Dataset`. While communications with TSA servers is designed to be resilient, there is a limit to what can (or should) be handled without application involvement.

When creating a `ClusteredDatasetManager`, a TCP connection to a server in each TSA stripe is opened by the client. The TCP connection between a TCStore client and a Terracotta server is a full-duplex, bidirectional channel between client and server. When establishing a connection, the TCStore client uses the `ClusteredDatasetManagerBuilder.withConnectionTimeout` value to limit the amount of time permitted for the initial connections with the servers to be set up. Once a connection is established, the health of the connection is periodically assessed independently by server and client.

Server-Side Connection Management

On the server side, a leasing mechanism is used. This is described in the section *Connection Leasing* in the *Terracotta Server Administration Guide*. When the client initially connects, a timed "lease" is granted by the server to that client. The lifetime of this lease is controlled by the `client-lease-duration` server configuration property. An active client is expected to renew this lease with the server before the lease expires. For TCStore clients, this happens without application involvement. If the lease is successfully renewed, normal operations continue for that client. If the client fails to renew the lease before it expires, the server considers the client dead and closes the sever side of the connection - undelivered responses for that client are discarded and client state is removed from the server. The client will eventually observe that the connection closed.

Client-Side Connection Management

In the TCStore client, three (3) connection health mechanisms are used: receipt of server responses to operations using the connection, a "connection health checker", and the client-side of the leasing mechanism used by the server.

I/O Operation Error Handling

If, during a read or write over the TCP connection to the server, the client experiences an error that does not indicate the TCP connection is intentionally closed, the client attempts to establish a new connection to the server (or a another configured peer in the stripe) within the scope of the current TCStore operation. From the application's point of view, the operation is not interrupted but just takes longer than usual. During this *reconnect phase*, connection attempts are repeated at

specified intervals and continue until (1) a connection is established or (2) the client's lease expires. *This level of reconnect is separate from the TCStore connection resiliency described below.*

During this reconnect phase, multiple connection attempts may be made. How many attempts are made and at what frequency is governed by internally established values. If the client's lease expires during the reconnect phase, attempts to reconnect are halted and TCStore connection resiliency capabilities (described below) come into play.

If the active server to which the client was connected fails and a former passive becomes active, the interval designated by the `client-reconnect-window` property is in force. A client establishing a connection to the new active server within (1) the time remaining in the client's lease **and** (2) the interval designated by `client-reconnect-window`, can resume operations without interruption. If either the client's lease or the `client-reconnect-window` expires, TCStore connection resiliency capabilities (described below) come into play.

Connection Health Checker

The "connection health checker" uses a "ping/response" mechanism during periods when the client is idle to ensure the client remains connected to the server. If the server does not respond to the pings, the server is considered (by the client) "unresponsive"; the client closes its side of the connection and the TCStore connection resiliency capabilities come into play.

Connection Leasing

A TCStore client also relies on the leasing mechanism. As described above, a lease is granted by the server and must be renewed by the client before the lease expires - within the interval specified by the `client-lease-duration` server configuration property. If lease renewal fails, the client considers the server unavailable and closes its side of the connection causing operations pending on that connection to be interrupted. At this point, the TCStore connection resiliency capabilities come into play.

TCStore Connection Resiliency

TCStore connection resiliency comes into action when an unrequested connection closure is observed on the client. This includes:

- lease expiration (described above) as observed in the client,
- connection closure forced by lease expiration on the server,
- connection rejection that occurs by a late reconnect attempt following a server fail-over (`client-reconnect-window` expiration), and
- network conditions that manifest as a closed connection.

The connection resiliency code suspends TCStore operations using the now-closed connection and attempts to reconnect with the cluster using alternate servers if necessary. While the time allowed for each connection attempt is controlled by the `ClusteredDatasetManagerBuilder.withConnectionTimeout` value, connection attempts are repeated until a connection is successfully established or the reconnection time limit (controlled by the

`ClusteredDatasetManagerBuilder.withReconnectTimeout` value) is exceeded. **Regardless of the withReconnectTimeout setting, at least one (1) reconnection attempt is made.**

By default, operations in TCStore wait for a reconnection FOREVER (`withReconnectTimeout` = 0) unless:

1. the connection is closed (by closing the associated `DatasetManager`) **OR**
2. the reconnection is interrupted (by interrupting the client application thread attempting the reconnection).

StoreOperationAbandonedException

If the client reconnects, suspended operations resume *with the exception of operations for which a server request was made prior to observing the connection closure*. For these "in-flight" operations, a `StoreOperationAbandonedException` is thrown to indicate the status of the operation is unknown. The application must take its own steps to determine if the operation completed, needs to be or can be repeated, or must be abandoned.

StoreOperationAbandonedException

Once a reconnection is made, operations awaiting the reconnection will either observe a `StoreOperationAbandonedException` or normal operation completion. Which of these is observed depends on what can be asserted (internally) about the state of the operation:

1. If a message has been presented to the server but has not been responded to, there is no way for the TCStore client code to determine if the operation message reached the server or, if it reached the server, the state of the operation initiated by that message. In this case, a `StoreOperationAbandonedException` is thrown.
2. If the operation was attempted while reconnect is underway, the operation will be retried (internally).

When a client receives a `StoreOperationAbandonedException`, it is *up to the client to determine whether or not the operation can be recovered* and, if so, what the recovery action must be. If application resilience is desired, the application **must** handle a `StoreOperationAbandonedException` which may be emitted from *any* TCStore operation that requires server interactions.

StoreReconnectFailedException

If the `withReconnectTimeout` time limit expires or the `DatasetManager` is closed while reconnecting, all operations suspended for that connection and any future operations against the affected `DatasetManager` are terminated with a `StoreReconnectFailedException`.

StoreReconnectFailedException

If a `StoreReconnectFailedException` is thrown, the affected server connection, the `DatasetManager` for which the connection was obtained, and any objects obtained from that `DatasetManager` are now effectively dead -- *the connection cannot be recovered and the DatasetManager is unusable*. If the client wishes to continue operations, the `DatasetManager` needs to be closed and a new `DatasetManager` instance obtained.

StoreReconnectInterruptedException

If the reconnecting thread is interrupted, that thread will observe a `StoreReconnectInterruptedException`; reconnection attempts will be picked up by another thread with a pending operation, if any.

StoreReconnectInterruptedException

A `StoreReconnectInterruptedException` is thrown if the client application thread under which the reconnect is being perform is interrupted using `Thread.interrupt()`. Unlike the `StoreReconnectFailedException`, the `DatasetManager` is not *yet* unusable - the reconnect procedure is picked up by another thread performing a TCStore operation against the affected `Dataset`. This interruption may be handled similarly to the `StoreOperationAbandonedException` - the interrupted operation is not *canceled*, it is simply no longer tracked - it may have completed and the response from the server just not arrived.

The `StoreOperationAbandonedException`, `StoreReconnectFailedException`, and `StoreReconnectInterruptedException` are *unchecked exceptions* (subclasses of the Java `RuntimeException`). Applications for which operational resilience is desired and that access a clustered Dataset need to handle at least the `StoreOperationAbandonedException` for any activity for which resilience is desired.

CRUD Operations

DatasetReader and DatasetWriterReader

A `DatasetReader` is required to read records from the dataset, and `DatasetWriterReader` allows add/update/delete operations on a dataset.

```
DatasetWriterReader<String> writerReader = persons.writerReader(); // <1>
```

1	`Dataset.writerReader` returns the required `DatasetWriterReader` for mutative access. Similarly, `Dataset.reader` returns a `DatasetReader`.

Adding Records

```
String person1 = "p1";
writerReader.add(person1,                    // 1
    Person.FIRST_NAME.newCell("Marcus"),     // 2
    Person.LAST_NAME.newCell("Aurelius"),
    Person.RATED.newCell(true),
    Person.NICENESS.newCell(0.65D));
```

1	The `DatasetWriterReader` API provides the `add` method which takes the specified key of the record,
2	And var-args of `Cell`. Another variant takes an `Iterable` of cells.

Accessing Records

```
Record<String> marcus =
    writerReader.get(person1).orElseThrow(AssertionError::new); // <1>
```

1	`DatasetReader` has a `get` method which takes a key as the argument. It returns an `Optional` of record. If the dataset doesn't have a record against the provided key `Optional.isPresent` will return false.

Update Existing Records

```
writerReader.update(marcus.getKey(),
    UpdateOperation.write(Person.NICENESS, 0.85D)); // <1>
writerReader.update(person2, UpdateOperation.allOf(
    UpdateOperation.write(Person.RATED, false),
    UpdateOperation.remove(Person.NICENESS))); // <2>
writerReader.update(person3, UpdateOperation.allOf(
    UpdateOperation.write(Person.RATED, true),
    UpdateOperation.write(Person.NICENESS, 0.92D)));
```

1	The `DatasetWriterReader.update` method requires the key of the record to be updated along with an `UpdateOperation` of the cell. The `UpdateOperation.write` method has overloaded variants which can be used to add/update cells in an existing record.
2	For updating multiple cells simultaneously, you can use `UpdateOperation.allOf` which takes a var-arg. To remove a cell use `UpdateOperation.remove`. Note that all these updates only happen to an existing record. If the record doesn't exist, an update operation will not result in the addition of a record against the provided key.

Deleting Records

```
writerReader.delete(marcus.getKey()); // <1>
```

1	`DatasetWriterReader.delete` takes key and returns `true` if the record deletion was successful.

Accessor APIs for CRUD

Another way to perform CRUD operations on a dataset is through using the `ReadWriteRecordAccessor` API. It provides more control over read-write operations on a record with mapping and conditional reads/writes.

```
ReadWriteRecordAccessor<String> recordAccessor = writerReader.on(person3); // <1>
recordAccessor.read(record -> record.get(Person.NICENESS).get()); // <2>
recordAccessor.upsert(Person.BIRTH_YEAR.newCell(2000),
    Person.PICTURE.newCell(new byte[1024])); // <3>
Optional<Integer> ageDiff = recordAccessor.update(UpdateOperation.write(
    Person.BIRTH_YEAR.newCell(1985)), (record1, record2) ->
    record1.get(Person.BIRTH_YEAR).get() -
```

```
        record2.get(Person.BIRTH_YEAR).get()); // <4>
ConditionalReadWriteRecordAccessor<String> conditionalReadWriteRecordAccessor =
    recordAccessor.iff(record ->
         record.get(Person.BIRTH_YEAR).get().equals(1985)); // <5>
Record<String> record = conditionalReadWriteRecordAccessor.read().get(); // <6>
conditionalReadWriteRecordAccessor.update(
    UpdateOperation.write(Person.RATED, false)); // <7>
conditionalReadWriteRecordAccessor.delete(); // <8>
```

1	An accessor for a record can be obtained by calling `DatasetWriterReader.on` which takes a key as the argument. `DatasetReader.on` returns a `ReadRecordAccessor` which has read only access to the record.
2	The `read` method takes a `Function` as an argument which maps the record to the required output.
3	The `upsert` method is like the same verb in a RDBMS: it will `add` if the record is absent or `update` if the record is present.
4	There is an advanced `update` that takes an additional `BiFunction` as mapper along with an `UpdateOperation`. The function maps the record that existed before the update and the record that resulted from the update.
5	Another variant allows conditional read/writes on the record. `ReadWriteRecordAccessor.iff` takes a predicate, the operations done on `ConditionalReadWriteRecordAccessor` are supplied with the same predicate. If the predicate returns `true`, the operation is executed against the record.
6	If the predicate returns `true`, the read will return a record.
7	The record will only be updated if the predicate returns `true`.
8	Similarly, the deletion succeeds if the predicate was `true`.

Please refer to the API documentation for more details.

Streams

Record Stream

A `RecordStream` is a `Stream<Record>` - a stream of `Record` instances. All operations defined in the Java 8 `Stream` interface are supported for `RecordStream`. Obtained using the `DatasetReader.records()` method, a `RecordStream` is the primary means of performing a query against a TCStore `Dataset`.

As with a `java.util.stream.Stream`, a `RecordStream` may be used only once. Unlike a Java `Stream`, a `RecordStream` closes itself when the stream is fully consumed through a terminal operation other than `iterator` or `spliterator`. (Even so, it is good practice to close a `RecordStream` using a try-with-resources block or `RecordStream.close`.) There are no provisions for concatenating two `RecordStream` instances while retaining `RecordStream` capabilities.

Most RecordStream intermediate operations return a RecordStream. However, operations which perform a transformation on a stream element may return a Stream<Record> which is **not** a RecordStream. For example, map(identity()) returns a Stream<Record> which is not a RecordStream.

Note:
In a clustered configuration, a stream pipeline formed against a RecordStream, in addition to being composed of intermediate and terminal operations (as described in the Java 8 package java.util.stream), is comprised of a server-side and a client-side pipeline segment. Every RecordStream originates in the server. As each operation is added during pipeline construction, an evaluation is made if the operation and its arguments can be run in the server (extending the server-side pipeline) - many pipeline operations can be run in the server. The first operation which cannot be run in the server begins the client-side pipeline. A stream pipeline may have both server-side and client-side pipeline segments, only a server-side segment, or only a client-side segment (other than the stream source). Each Record or element passing through the stream pipeline is processed first by the server-side pipeline segment (if any) and is then passed to the client-side pipeline segment (if the client-side pipeline segment exists) to complete processing.

The following code creates a RecordStream and performs few operations on the records of the stream:

```
long numMaleEmployees = employeeReader.records()   // <1>
    .filter(GENDER.value().is('M'))   // <2>
    .count();   // <3>
```

1	The DatasetReader.record() method returns a RecordStream delivering Record instances from the Dataset referred to by the DatasetReader.
2	Stream intermediate operations on a RecordStream return a stream whose type is determined by the operation and its parameters. In this example, filter provides a RecordStream.
3	A Stream terminal operation on RecordStream produces a value or a side-effect. In this case, count returns the number of Record instances passing the filter above.

Additional operations supported On RecordStream

```
Optional<Record<Integer>> record = employeeReader.records()
  .explain(System.out::println)  // <1>
  .batch(2) // <2>
  .peek(RecordStream.log("{} from {}", NAME.valueOr(""),
      COUNTRY.valueOr(""))) // <3>
  .filter(COUNTRY.value().is("USA"))
  .findAny();
long count = employeeReader.records()
    .inline()  // <4>
    .count();
```

1	The RecordStream.explain operation observes the stream pipeline and provides the pre-execution analysis information for this stream pipeline. It takes, as a parameter, a Consumer which is passed an explanation of the stream execution plan. RecordStream.explain returns a RecordStream for further pipeline construction. For explain

	to be effective, the pipeline must be terminated - the plan is not determined until the pipeline begins execution. The `explain Consumer` is called once the pipeline is closed. For a `RecordStream` against a clustered TCStore configuration, the explanation identifies the operations in each of the server-side and client-side pipeline segments.
2	In a clustered configuration, a `RecordStream` *batches* elements transferred from the server to the client, when possible, to reduce latencies involved in network transfer. The number of records or elements returned to the client at one time can be influenced using the `RecordStream.batch` operation. The `batch` operation takes a batch size as parameter and uses it as a hint for the batch size to use when transferring elements. `RecordStream.batch` returns a `RecordStream` for further pipeline construction. **Note:** When batching is **not** disabled, multiple elements may complete processing in the server-side pipeline segment before any elements are presented to the client-side pipeline segment. If one-at-a-time element processing is required, the `inline` operation (described below) must be used to disable batching.
3	The `RecordStream.log` method produces a `Consumer` for use in `Stream.peek` to log a message according to the specified format and arguments. The first argument provides a message format like that used in the SLF4J `MessageFormatter.arrayFormat` method. Each subsequent argument supplies a value to be substituted into the message text and is a mapping function that maps the stream element to the value to be substituted. The formatted message is logged using the logging implementation discovered by SLF4J (the logging abstraction used in TCStore). If the `peek(log(…))` operation is in the server-side pipeline segment, the formatted message is logged on the TCStore server. If the `peek(log(…))` operation is in the client-side segment, the formatted message is logged in the client.
4	The `RecordStream.inline` operation disables the element batching discussed above. When `inline` is used, each stream element is processed by both the server-side and client-side pipeline segments before the next element is processed. `RecordStream.inline` returns a `RecordStream` for further pipeline construction.

Mutable Record Stream

Obtained from the `DatasetWriterReader.records()` method, a `MutableRecordStream` extends `RecordStream` to provide operations through which `Record` instances in the `RecordStream` may be changed. No more than one of the mutating operations may be used in a pipeline. The changes in a `Record` from a `MutableRecordStream` mutation operation affect only the `Dataset` from which `MutableRecordStream` was obtained (and to which the `Record` belongs).

The following are the operations added in `MutableRecordStream`:

mutateThen

The `MutableRecordStream.mutateThen` operation is an intermediate operation that accepts an `UpdateOperation` instance describing a mutation to perform on every Record passing through the `mutateThen` operation. The output of `mutateThen` is a `Stream<Tuple<Record, Record>>` where the `Tuple` holds the *before* (`Tuple.first()`) and *after* (`Tuple.second()`) versions of the `Record`.

```
double sum = employeeWriterReader.records()            // 1
    .mutateThen(UpdateOperation.write(SALARY).doubleResultOf(
        SALARY.doubleValueOrFail().increment()))              // 2
    .map(Tuple::getSecond)                                    // 3
    .mapToDouble(SALARY.doubleValueOrFail())
    .sum();
```

1	The `DatasetWriterReader.record()` method, not `DatasetReader.record()`, returns a `MutableRecordStream` which is a `Stream` of Records of the `Dataset` referred by the `DatasetWriterReader`.
2	`MutableRecordStream.mutateThen()` is an intermediate operation and takes in `UpdateOperation` as parameter and performs the update transformation against the `Record` instances in the stream.
3	`MutableRecordStream.mutateThen()` returns a `Stream` of new `Tuple` instances holding before and after `Record` instances. Note that it does not return a `RecordStream` or a `MutableRecordStream`.

deleteThen

The `MutableRecordStream.deleteThen` operation is an intermediate operation that deletes all `Record` instances passing through the `deleteThen` operation. The output of `deleteThen` is a `Stream<Record>` where each element is a deleted `Record`. (Note the output is neither a `RecordStream` nor a `MutableRecordStream`.)

```
employeeWriterReader.records()
    .filter(BIRTH_YEAR.value().isGreaterThan(1985))
    .deleteThen()   // <1>
    .map(NAME.valueOrFail())   // <2>
    .forEach(name -> System.out.println("Deleted record of " + name));
```

1	`MutableRecordStream.deleteThen()` is an intermediate operation and deletes every `Record` in the stream.
2	`MutableRecordStream.deleteThen()` returns a `Stream` of the deleted `Record` instances. Note that it does not return a `RecordStream` or a `MutableRecordStream`.

mutate

The `MutableRecordStream.mutate` operation is a terminal operation that accepts an `UpdateOperation` instance describing a mutation to perform on every `Record` reaching the `mutate` operation. The return type of the `mutate` operation is `void`.

```
employeeWriterReader.records()
    .filter(GENDER.value().is('M'))
    .mutate(UpdateOperation.write(SALARY)
        .doubleResultOf(SALARY.doubleValueOrFail().decrement())); <1>
```

1	`MutableRecordStream.mutate()` takes in `UpdateOperation` as parameter and performs the update transformation against the`Record` instances in the stream. This is a terminal operation and returns nothing.

delete

The `MutableRecordStream.delete` operation is a terminal operation deletes every `Record` reaching the `delete` operation. The return type of the `delete` operation is `void`.

```
employeeWriterReader.records()
    .filter(BIRTH_YEAR.value().isLessThan(1945))
    .delete(); // <1>
```

1	`MutableRecordStream.delete()` deletes every `Record` in the stream. This is a terminal operation and returns nothing.

Stream pipeline execution and concurrent record mutations

During stream pipeline execution on a `Dataset`, concurrent mutation of records on it are allowed. Pipeline execution does not iterate over a point in time snapshot of a `Dataset` - changes by concurrent mutations on a `Dataset` may or may not be visible to a pipeline execution depending on the position of its underlying `Record` iterator.

Stream pipeline portability

In a clustered configuration, the `Record` instances accessed through a `RecordStream` are sourced from one or more Terracotta servers. For large datasets, this can involve an enormous amount of data transfer. To reduce the amount of data to transfer, there are `RecordStream` capabilities and optimization strategies that can be applied to significantly reduce the amount of data transferred. One of these capabilities is enabled through the use of *portable pipeline operations*. This capability and others are described in the section "Streams" on page 26.

Asynchronous API

TCStore provides an asynchronous API based around the Java 9 `CompletionStage` API.

```
AsyncDatasetWriterReader<String> asyncAccess = counterAccess.async(); // <1>
Operation<Boolean> addOp = asyncAccess.add("counter10", counterCell.newCell(10L)); //
 <2>
Operation<Optional<Record<String>>> getOp =
    addOp.thenCompose((b) -> asyncAccess.get("counter10")); // <3>
Operation<Void> acceptOp = getOp.thenAccept(or -> or.ifPresent( // <4>
    r -> out.println("The record with key " + r.getKey() + " was added")));
try {
  acceptOp.get();   // <5>
} catch (InterruptedException | ExecutionException e) {
  e.printStackTrace();
}
```

1	The asynchronous API is accessed through the `async()` method on an existing reader or writer-reader.
2	Methods then create asynchronous executions represented by `Operation` instances.

3	Operations can be composed with other operations using the usual `CompletionStage` methods.
4	Operations can also be composed with synchronous operations still yielding `Operation` instances.
5	`Operation` also extends `Future` for easy interoperability with synchronous code.

Note:

The current asynchronous implementation is a simple thread-pool based skin over the synchronous API. It is not currently interacting optimally with the asynchronous nature of the underlying Terracotta platform.

Functional DSL

The functional DSL exists to allow users to express arguments passed to TCStore operations in a form that is both portable between clients and servers (over the network), and whose underlying behavior can be introspected and understood by the TCStore software. DSL expressions are the preferred form for **all** functional arguments passed to TCStore.

Cell Operations

Functional operations on cells and their associated values can be created via references to the associated cell definition objects.

```
BoolCellDefinition definition = defineBool("cell");
Predicate<Record<?>> exists = definition.exists(); // <1>
Predicate<Record<?>> isTrue = definition.isTrue(); // <2>
```

1	A cell existence predicate. The predicate returns true if the passed record contains a cell of this definition. This is available for all definition types.
2	A boolean cell value predicate. The predicate returns true if the passed record contains a cell of this definition with a true value (this means an absence of the cell results in a false value).

The types returned from the DSL are fluent builders so you can derive functions from existing functions.

```
StringCellDefinition definition = defineString("cell");
BuildableComparableOptionalFunction<Record<?>, String>
    value = definition.value(); // <1>
Predicate<Record<?>> isFoo = value.is("foo"); // <2>
Predicate<Record<?>> isAfterBar = value.isGreaterThan("bar"); // <3>
```

1	A cell value extraction function. This is a subtype of `Function<Record<?>, Optional<String>>` but can also be built upon.

2	A value equality predicate. The predicate returns true if the passed record contains a cell of this definition whose value is "foo".
3	An open range predicate. The predicate returns true if the passed records contains a cell of this definition whose value is strictly greater than "bar".

The available build methods are specialized to the type of the cell in question. Numerically typed cells can be used to do numeric manipulations.

```
IntCellDefinition definition = defineInt("cell");
BuildableToIntFunction<Record<?>> intValue = definition.intValueOr(0);  // <1>
BuildablePredicate<Record<?>> isGreaterThanOrEqualTo4 =
    intValue.isGreaterThanOrEqualTo(4);                                 // <2>
ToIntFunction<Record<?>> incremented = intValue.increment();            // <3>
Comparator<Record<?>> comparator = intValue.asComparator();             // <4>
```

1	An integer extracting function that returns a specialized builder type, that is also a primitive int bearing function.
2	A numeric open range predicate. Ranges are available for all comparable cell types.
3	An integer extracting function that outputs the value incremented (+1).
4	A comparator over extracted values.

Cell derived expressions will be frequently used as:

- Predicates for streams and CRUD operations.
- Mappers for streams and read operations.
- Input for functional update operations.

Complex Data Types (CDTs)

`LIST` and `MAP` typed cells have additional DSL concepts. Lists and Maps are not *scalar* values, they contain multitudes. This affects their interaction with the DSL in a couple of ways. First, they have three common DSL predicates that generate scalar values and thus integrate directly with the existing DSL: `isEmpty()`, `notEmpty()`, `size()`.

For example, this fragment shows conditional filtering on a `LIST` cell.

```
ListCellDefinition listDef=...
...stream().filter(listDef.value().size().greaterThan(10)).
```

The `LIST` cell definition's `value()` family of methods use the `BuildableListFunction` hierarchy and add `contains()` to test if the list contains a specified `TypedValue`. The `MAP` cell definition's `value()` family of methods add `containsKey()` and `containsValue()`.

Complex Data Type Content Selection

Querying inside CDTs requires using the `find()` method on all CDTs. This is accessible in the DSL via the corresponding `find()` family of functions on `LIST` or `MAP` cell definitions, or `Record.find()` if you wish to query on the entire `Record`.

These `find()` methods return variants of `BuildableTypedValueSelectionFunction`. This hierarchy allows further refinement of the resulting Selection via types, etc.

If at any point the DSL is used to restrict the subset to a typed scalar value, then that builder can be used with any of the existing DSL to construct possibly portable filters.

Update Operations

Update operation instances are used to express mutation used in either single-key update operations, or against stream contents via a `MutableRecordStream` operation. Update operations are created via static accessor methods on the `UpdateOperation` class

```
IntCellDefinition defnA = defineInt("cell-a");
IntCellDefinition defnB = defineInt("cell-b");
UpdateOperation<Long> install =
    UpdateOperation.install(defnA.newCell(42), defnB.newCell(42));  // 1
UpdateOperation.CellUpdateOperation<Long, Integer> write =
    UpdateOperation.write(defnA).value(42);                         // 2
UpdateOperation.CellUpdateOperation<Long, Integer> increment =      // 3
    UpdateOperation.write(defnA)
        .intResultOf(defnA.intValueOr(0).increment());
UpdateOperation.CellUpdateOperation<Long, Integer> copy =
    UpdateOperation.write(defnB).intResultOf(defnA.intValueOr(42));
UpdateOperation<Long> aggregate =
    UpdateOperation.allOf(increment, copy);                         // 4
```

1	Install a specific list of cells. An *install* operation replaces all existing cells.
2	Write an individual cell. This will overwrite an existing cell or create a new one as necessary.
3	Write an individual cell with a value given by executing the given function against the current record.
4	Perform a list of individual cell updates as a single unit.

Update Output

Update operations output a pair of values representing the state before and after the mutation application. This is either in the form of a pair of values passed to a bi-function or as a tuple of records.

```
BiFunction<Record<?>, Record<?>, Integer> inputBiFunction =
    UpdateOperation.input(defnA.valueOr(42)); // <1>
BiFunction<Record<?>, Record<?>, Integer> outputBiFunction =
    UpdateOperation.output(defnA.valueOr(42)); // <2>
Function<Tuple<Record<?>, ?>, Integer> inputTupleFunction =
    Tuple.<Record<?>>first().andThen(defnA.valueOr(42)); // <3>
```

```
Function<Tuple<?, Record<?>>, Integer> outputTupleFunction =
    Tuple.<Record<?>>second().andThen(defnA.valueOr(42)); // <4>
```

1	Extract the input value of `cell-a` from the resultant bi-function's two arguments.
2	Extract the output value of `cell-a` from the resultant bi-function's two arguments.
3	Extract the value of `cell-a` from the first value of the resultant function's tuple argument.
4	Extract the value of `cell-a` from the second value of the resultant function's tuple argument.

Both tuple and bi-function forms follow the convention that input records are the first argument or tuple member, and output records are the second argument or tuple member.

Collectors

To support stream collection operations a mirror of the JDK `java.util.stream.Collectors` class that creates collectors transparent to TCStore at `com.terracottatech.store.function.Collectors`.

Indexes

The records stored in a dataset are accessed for CRUD operations using the key against which the record is held. However, for stream queries there is an option to use secondary indexes for better query performance. Secondary indexes can be created on a specific `Cell`, thus all the records having that cell will be indexed. The queries on the indexed cell will try to use the index for optimized results.

Creating Secondary Indexes

The code snippet provided below depicts how to create/destroy indexes.

```
DatasetManager datasetManager =
    DatasetManager.clustered(clusterUri).build();
DatasetConfiguration configuration = datasetManager.datasetConfiguration()
  .offheap(offHeapResourceId)
  .index(CellDefinition.define("orderId", Type.STRING),
       IndexSettings.BTREE)                              // 1
  .build();
datasetManager.newDataset("indexedOrders", Type.LONG, configuration);
Dataset<Long> dataset =
    datasetManager.getDataset("indexedOrders", Type.LONG);
Indexing indexing = dataset.getIndexing();              // 2
Operation<Index<Integer>> indexOperation = indexing.createIndex(
   CellDefinition.define("invoiceId", Type.INT),
       IndexSettings.BTREE);                            // 3
Index<Integer> invoiceIdIndex = indexOperation.get();    // 4
```

1	An `Index` can be created while the dataset is being created. The `DatasetConfigurationBuilder.index` method takes a `CellDefinition` and an `IndexSettings`. Currently only `IndexSettings.BTREE` is supported for secondary indexes.

2	In case there is a need to index a cell after dataset is created, that can be done as well. For that, `Indexing` is provided by `Dataset.getIndexing` to create/delete indexes on a dataset.
3	The `Indexing.createIndex` method again takes a `CellDefinition` and an `IndexSettings`, to return an `Operation` of `Index`. `Operation` represents the asynchronous execution of the long running indexing operation.
4	You get an `Index` when the operation completes.

Getting Index Status

The code snippet depicts how to determine the status of indexes.

```
Collection<Index<?>> allIndexes = indexing.getAllIndexes();    // 1
Collection<Index<?>> liveIndexes = indexing.getLiveIndexes();  // 2
```

1	`Indexing.getAllIndexes` returns all the indexes created on the dataset, regardless of their status.
2	`Indexing.getLiveIndexes` returns only those indexes whose `Status` is `LIVE`.

Destroying Indexes

The code snippet depicts how to determine the status of indexes.

```
indexing.destroyIndex(invoiceIdIndex);  // 1
```

1	An existing `Index` can be destroyed using `Indexing.destroyIndex`.

Indexes in HA setup

Creating an index is a long running operation. With an HA setup, indexes are created asynchronously on the mirrors. This implies that if an index creation has completed and the status is LIVE, the index creation might still be in progress on mirrors which might complete eventually. Also when a new mirror comes up, the records on the active are synced to mirror, but they are indexed only when syncing of data is complete. Thus indexing on a new mirror is done asynchronously.

Please refer to API documentation for more details.

2 Usage and Best Practices

Stream Optimizations

When performing queries or mutations using a stream pipeline on a `RecordStream` or `MutableRecordStream` (referred to collectively in this documentation as `RecordStream`) there are several ways a user can influence the performance of the pipeline. The primary methods, using pipeline portability and cell indexes, are described in the sections "Pipeline Portability" on page 38 and "Index Use" on page 43. There is also a tool, the *stream plan*, that provides visibility on the effectiveness of the performance methods; this is described in the followng section "Stream Plan" on page 38.

Stream Plan

There is a tool available to help a user understand how a pipeline based on a `RecordStream` will be executed - the *stream plan*. It is observed using the object presented to the `RecordStream.explain(Consumer<Object>)` pipeline operation. This object represents the system's understanding of the pipeline and includes information about how the pipeline will be executed by TCStore.

Note:

The plan object is **not** a programmatic API. The object is intended to be converted to `String` using the `toString()` method and reviewed by a human. The content and format of the `String` are subject to change without notice.

Looking at the plan, a user can determine:

1. what portions of the pipeline are portable and may be executed on the server;
2. what portions of the pipeline are non-portable and must be executed on the client;
3. what index, if any, is used for data retrieval.

Sample plans are included in the discussions below.

In a striped TCStore configuration, multiple plans are included in the output - one (1) for each stripe in the configuration. Each server in a stripe will calculate a stream plan based on state extant in that server so plans may differ from stripe to stripe.

The stream plan for a pipeline is provided to the `explain Consumer` only after the pipeline completes execution and the stream is closed. (This is, in part, due to the fact that the stream plan is not computed until the pipeline begins execution - that is, once the terminal operation is appended to the pipeline.)

Pipeline Portability

As discussed in the section "Record Stream" on page 26, `RecordStream` pipelines in a TCStore clustered configuration are split into server-side and client-side segments. The best performing TCStore stream pipelines are those which limit the amount of data transferred between the server and the client. In general, the more processing that can be performed in the server - close to the data - the better.

For an operation to be run in the server, the operation and its arguments must be *portable*. A portable operation is one for which the operation, its context and its arguments are understood through introspection. This introspection is enabled by the use of the TCStore Functional DSL (see the section "Functional DSL" on page 31). Most, but not all, function instances produced from the DSL are portable.

Note:

Operations using *lambda* expressions ("arrow" operator) or *method reference* expressions (double colon separator) are **not** portable and must be executed in the client.

Every `RecordStream` pipeline begins as a portable pipeline - the stream's data source is the server. As each operation is added to the pipeline, that operation and its arguments are evaluated for portability - in general, if the arguments (if any) provided to the operation are produced using the TCStore DSL, the operation will be portable. (Exceptions are noted in "DSL Support for Portable Operations" on page 41 below.) The portable, server-side pipeline segment is extended with each portable operation appended to the pipeline. The non-portable, client-side pipeline segment begins with the first non-portable operation and continues through to the pipeline's terminal operation.

Note:

Even if an otherwise portable operation is appended to the pipeline *after* a non-portable operation, that otherwise portable operation is executed on the client - the stream elements are already being transferred from the server to the client.

To determine how much of a pipeline is portable, use the `RecordStream.explain(Consumer<Object>)` operation. This makes a *stream plan* available which may be used to determine what portions of a pipeline are portable. Stream plans are introduced in *Stream Plan* section above.

Note:

The `explain` operation does not affect pipeline portability - `explain` is a *meta-operation* and sets an observer for the stream plan but does not actually add an operation to the pipeline.

The `peek` operation *can* affect pipeline portability. If the `Consumer` provided to the `peek` operation is non-portable, the pipeline segment beginning with that `peek` operation will be rendered non-portable and forced to run on the client. A warning is logged if a non-portable `peek` is appended to a pipeline that, to that point, is portable. The `RecordStream.log` method can be used to produce a portable `Consumer` for peek.

Examples

In the examples that follow, the following definitions are presumed:

```
import static java.util.stream.Collectors.toList;
public static final StringCellDefinition TAXONOMIC_CLASS =
    defineString("class");
RecordStream recordStream = dataset.records();
```

Non-Portable Operations

This example shows a pipeline using an operation with a non-portable argument - a lambda expression - making the operation non-portable. In this example, all records in the dataset are shipped to the client for processing by the filter operation.

Non-Portable Pipeline:

```
List<String> result = recordStream
    .explain(System.out::println)
    .filter(r -> r.get(TAXONOMIC_CLASS).orElse("").equals("mammal"))  // 1
    .map(TAXONOMIC_CLASS.valueOrFail())     // 2
    .collect(toList());
```

1	Using Java *lambda expressions* (expressions using the "arrow" operator) **always** produce non-portable operations.
2	The `map` operation in this example *could* be portable but is not because it follows a non-portable operation - once a non-portable operation is used and pipeline execution shifts to the client, subsequent operations are made non-portable.

Stream Plan - No Portable Operations:

```
Stream Plan
  Structure:
    Portable:
       None      // 1
    Non-Portable:
       PipelineOperation{FILTER(com.terracottatech.store.server.
           RemoteStreamTest$$Lambda$504/1753714541@51bf5add)}  // 2
       PipelineOperation{MAP(class.valueOrFail())}
       PipelineOperation{COLLECT_1(
           java.util.stream.Collectors$CollectorImpl@7905a0b8)}
  Server Plan: 0970e486-484c-4e04-bb8e-5fe477d47c0d
    Stream Planning Time (Nanoseconds): 2611339
    Sorted Index Used In Filter: false
    Filter Expression: true
    Unknown Filter Count: 0
    Unused Filter Count And Filters (If Any): 0
    Selected Plan: Full Dataset Scan   //3
```

1	No portable operations are identified.
2	Several non-portable operations are identified. These operations are all executed in the client.
3	Pipelines having no portable operations require a full dataset scan for data retrieval.

Portable Operations

This example shows a pipeline expressing the same sequence of operations as the previous example but using portable operation arguments making the majority of the pipeline portable. Unlike the previous example, both filtering and mapping are performed *on the server* limiting what is transferred to the client to that data that actually needs to be collected.

Portable Pipeline:

```
List<String> result = recordStream
   .explain(System.out::println)
   .filter(TAXONOMIC_CLASS.value().is("mammal"))  // 1
   .map(TAXONOMIC_CLASS.valueOrFail())            // 2
   .collect(toList());
```

1	This `filter` operation expresses the same selection criterion as the first example but does so using a *portable* DSL expression.
2	Unlike the first example, the `map` operation in this pipeline is portable - all preceding operations in the pipeline are portable so the `map` operation can be portable.

Stream Plan - Portable Operations:

```
Stream Plan
  Structure:
     Portable:
        PipelineOperation{FILTER((class==mammal))}  // 1
        PipelineOperation{MAP(class.valueOrFail())}
     Non-Portable:
         PipelineOperation{COLLECT_1(
            java.util.stream.Collectors$CollectorImpl@1e13529a)}    // 2
  Server Plan: ecc2db4d-1da7-4822-ad8a-b2f469fce4d5
     Stream Planning Time (Nanoseconds): 99065863
     Sorted Index Used In Filter: false
     Filter Expression: (class==mammal)
     Unknown Filter Count: 0
     Unused Filter Count And Filters (If Any): 0
     Selected Plan: Full Dataset Scan     // 3
```

1	Two (2) portable operations are identified.
2	One (1) non-portable operation is identified. This operation, the `toList` collector, must be run in the client.
3	Pipelines using portable operations may use an index-based data retrieval if an index is available. In this example, no index for the `class` (`TAXONOMIC_CLASS`) cell was defined. See the section "Index Use" on page 43 below.

DSL Support for Portable Operations

As discussed in the section "Functional DSL" on page 31, the DSL methods permit expression of pipeline operation arguments in a manner which can be portable between client and server. However, as a growth point in TCStore, the DSL methods may produce non-portable expressions as well.

A method in the DSL produces an instance of one of the interfaces found in `java.util.function` - `Predicate`, `Function`, `Consumer`, `BiFunction`, `ToDoubleFunction`, `ToIntFunction`, `ToLongFunction`, etc. - or found in `java.util.stream` like `Collector`. For the instance to be portable, the instance must be from a TCStore implementation that is designed and implemented to be portable. There

are currently no provisions for a user to extend the collection of portable operations by implementing their own portable DSL extensions.

The following is a list of the DSL methods that produce *non-portable* expressions:

- **UpdateOperation.custom** The `UpdateOperation.custom` method is intended to provide a means of performing updates too complex to be expressed using the other `UpdateOperation` methods - `custom` is not intended to be used for portable operations so it will not produce a portable function instance.
- **Collectors Methods** The following `com.terracottatech.store.function.Collectors` methods return non-portable `Collector` implementations:

`averagingDouble`	`groupingBy`	`partitioningBy`
`averagingInt`	`groupingByConcurrent`	`summingDouble`
`averagingLong`	`mapping`	`summingInt`
`composite`	`maxBy`	`summingLong`
`counting`	`minBy`	`varianceOf`
`filtering`		

  A `collect` operation, even when using a portable `Collector`, will partially execute in the client to perform result aggregation over the stripes in a multi-stripe configuration. A `collect` operation involving a `Collector` that *does not* perform a data reduction or aggregation operation will always involve data transfer to and execution in the client.

- **Comparator Methods** The `asComparator` method from the value accessors (`value()`, `doubleValueOrFail()`, etc.) on each of the `CellDefinition` subtypes and from `Record.keyFunction()` produce `Comparator` implementations that do not provide a portable implementation of the `thenComparing`, `thenComparingDouble`, `thenComparingInt`, or `thenComparingLong` methods.
- **Function.andThen / Consumer.andThen** Several of the DSL functions produce a specialized type of the `Function` or `Consumer` interfaces. Most of these specialized types do not implement the `andThen` method - the `andThen` method does not produce a portable instance. For example, `definition.value().andThen(Function)` where `definition` is a `CellDefinition` (or one of its subtypes) produces a non-portable instance even if the argument to `andThen` is portable.
- **Function.compose** Several of the DSL functions produce a specialized type of the `Function` interface. Most of these specialized types do not implement the `compose` method - the `compose` method does not produce a portable function instance. For example, `definition.value().compose(Function)` where `definition` is a `CellDefinition` (or one of its subtypes) produces a non-portable instance even if the argument to compose is portable.
- **multiply / divide** The type-specific value accessors on the numeric `CellDefinition` subtypes, for example `DoubleCellDefinition`.doubleValueOrFail(), each provide `multiply` and `divide` methods that produce a non-portable function instance.

- **length / startsWith** The value accessors of `StringCellDefinition` - `value()` and `valueOrFail()` - provide `length` and `startsWith` methods that produce a non-portable function instance.
- **All `StoreMap` / `StoreList` / `ComplexRecord`** All the DSL operations on Complex Data Types which involve the contents of the Complex Data Type are not currently portable; it is anticipated this will change in the future.

The number of DSL methods *and* the number of methods producing portable expressions will be extended over time.

Index Use

In combination with pipeline portability, `Predicates` used in `RecordStream.filter` operations used in the portable, server-side segment of the pipeline are analyzed for expressions referring to `CellDefinitions` on which an index is defined. Analysis by the server chooses one index through which the dataset is accessed to provide the `Record` instances for the stream. Because a TCStore index tracks only `Record` instances *having* the indexed `Cell`, `Record` instances without a value for the indexed `Cell` are not presented to the stream when an index is used.

Note:

Because an index provides only `Record` instances having the indexed cell, the `Predicate` analysis looks for uses of the `CellDefinition.value()` method. The other forms of value reference (`valueOr`, `valueOrFail`, `longValueOr`, `longValueOrFail`, etc.) are not supported in determining index use. So, while `TAXONOMIC_CLASS.value()` is considered for index use, `TAXONOMIC_CLASS.valueOrFail()` is not.

The analysis also includes a determination of whether or not a range query can be performed. The use of range comparisons (`value().isGreaterThan()`, `value().isLessThanOrEqualTo())` permits selection of a subset of the indexed `Record` instances using the index.

Example

For example, using the portable example from the section *Portable Operations* above, if an index is defined over the `TAXONOMIC_CLASS` `CellDefinition`, an index will be used when supplying `Record` instances to the pipeline.

Portable Pipeline:

```
List<String> result = recordStream
    .explain(System.out::println)
    .filter(TAXONOMIC_CLASS.value().is("mammal"))   // 1
    .map(TAXONOMIC_CLASS.valueOrFail())
    .collect(toList());
```

1	`TAXONOMIC_CLASS` is a reference to a `StringCellDefinition` over which an index is defined.

Stream Plan - Portable Operations & Using an Index

```
Stream Plan
  Structure:
    Portable:
```

```
      PipelineOperation{FILTER((class==mammal))}    // 1
      PipelineOperation{MAP(class.valueOrFail())}
  Non-Portable:
      PipelineOperation{COLLECT_1(
          java.util.stream.Collectors$CollectorImpl@1b410b60)}
Server Plan: a9c4a05c-7303-440c-90b5-d56bf518b66f
  Stream Planning Time (Nanoseconds): 138369229
  Sorted Index Used In Filter: true
  Filter Expression: (class==mammal)
  Unknown Filter Count: 0
  Unused Filter Count And Filters (If Any): 0
  Selected Plan: Sorted Index Scan                  // 2
    Cell Definition Name: class
    Cell Definition Type: String
    Index Ranges: (Number of Ranges = 1)
           Index Range 1: Range = mammal ::: Operation = EQ
```

1	As with the previous example, the same two (2) operations are portable. The `filter Predicate` refers to a `CellDefinition` over which an index is defined.
2	A "Sorted Index Plan" was chosen. The attributes of the access (`CellDefinition` information and the type of index query) are described.

Failover Tuning for TCStore

When setting up a high availability (HA) Terracotta Server Array (TSA) supporting TCStore datasets, the choice made for the `failover-priority` must be considered with TCStore in mind.

As discussed in the section *Choosing Consistency versus Availability* of the *Terracotta Server Administration Guide*, for the typical TCStore use case, `failover-priority` should be set to `consistency`.

Connection Pooling

Overview

The use of *connection pooling* is common among applications accessing relational databases through the Java Database Connectivity (JDBC) API.

Through a connection pool, application threads *borrow* a connection to the database for the duration of some unit of work and then return the connection to the pool for use by another application thread. This is done to avoid the overhead of establishing the connection to the database for each unit of work the application performs. A connection pool is frequently used by an application deployed as a servlet in a servlet engine with each servlet request performing a single unit of work.

So why not use a single JDBC connection for all requests? Technically, each `java.sql.Connection` implementation is thread-safe but many, if not most, implementations achieve this thread-safety through method synchronization - effectively single-threading operations using a single connection. Perhaps more importantly, JDBC transactions are scoped with the `Connection` - a `commit` by any thread using a `Connection` commits all activity using that `Connection`. So, sharing a `Connection`

among application threads requires the application to coordinate its work and tolerate the single-threaded processing of its requests.

While TCStore doesn't expose a connection object through its API, there are benefits to sharing some of the API objects among application threads. Among the TCStore API objects which should be considered for sharing are the `DatasetManager` and `Dataset` objects. Most TCStore API objects and methods are thread-safe without resorting to high-level synchronization. Under TCStore, each mutation performed on a `Dataset` is atomic and committed individually so the need for separate TCStore "connections" to address operation atomicity is not applicable.

`DatasetManager`

Note:
Obtaining a new `ClusteredDatasetManger` instance for each application unit of work will result in poor application performance.

A `DatasetManager` instance is as close as TCStore comes to having a connection object. A `DatasetManager` is the object through which an application gains access to and manages TCStore datasets. To interact with datasets residing in a TSA, an application needs a `ClusteredDatasetManager` instance. Creating a `ClusteredDatasetManager` instance (using `DatasetManager.clustered(uri).build()`) is a fairly expensive operation involving the creation of TCP connections (at least two per stripe) along with several exchanges between client and servers. Fortunately, a `ClusteredDatasetManager` holds no state related to operations against the dataset manager or datasets it manages - it is safe to share among application threads.

From the perspective of the Terracotta Management Console (TMC) each `ClusteredDatasetManager` instance is a client. If you require more granular visibility into your application thread operations, you should consider using separate `ClusteredDatasetManager` instances that correspond to the required granularity.

In addition to the expense of creating a `ClusteredDatasetManager` instance, most of the methods on a `ClusteredDatasetManager` instance are also fairly expensive to perform and should not be performed frequently. These operations are expensive not only in the amount of time required to complete the operations but on the impact on overall server performance.

Given the client and server resources consumed by a `ClusteredDatasetManager` instance, the instance should be closed when it's no longer needed. But, if a `DatasetManager` instance is shared among application threads, the `DatasetManager.close()` method should not be invoked unless and until all operations on `Dataset` instances obtained from the `DatasetManager` instance are complete - calling `close` may abruptly terminate in-progress operations.

`Dataset`

Note:
Using `DatasetManager.getDataset` to obtain reference to a `Dataset` for each application unit of work will result in poor application performance.

The TCStore `Dataset` object is the application's entry point to reading from, writing to, and managing indexes on a dataset. An application creates a dataset using a call to `DatasetManager.newDataset(…)` and obtains a reference to a previously created dataset using `DatasetManager.getDataset(…)`. As

mentioned above, each of these operations is somewhat time consuming and should not be done frequently. And, once created, a persistent dataset cannot be created again until it is destroyed so the new `Dataset` operation need not be repeated routinely.

To gain access to an already-created dataset, use the `DatasetManager.getDataset(…)` method. Again, this method is expensive and should not be repeated for every application unit of work. As with a `ClusteredDatasetManager` instance, a `Dataset` instance holds no state related to operations so it's safe to share `Dataset` instances among application threads.

Compared with `DatasetManager.getDataset(…)`, the methods on a `Dataset` instance are relatively inexpensive and can be performed in each application unit of work. However, the `Indexing` instance returned by the `Dataset.indexing()` method **should not** be used for routine operations. Creating an index and deleting an index are potentially expensive operations - applications should notadd an index to a dataset, perform processing using that index, and then remove that index.

Maintaining a client-side reference to a `Dataset` is not without server-side cost. As with a `ClusteredDatasetManager`, a `Dataset` instance should be closed when no longer needed. Again like a `ClusteredDatasetManager`, if a `Dataset` instance is shared among application threads, the `Dataset.close()` method should not be invoked unless and until all operations on that `Dataset` instance are complete - calling `close` may abruptly terminate in-progress operations.

Other TCStore Objects

In addition to the objects mentioned above, there are many other objects in the TCStore API. While these objects generally need not be part of pooling strategies, their existence must be taken into account when considering a pooling strategy. For example, each of these objects is derived directly or indirectly from a `Dataset` instance - when using a pooled `Dataset` instance, operations on these objects **must be complete** before returning the `Dataset` instance to the pool. No references to any object obtained directly or indirectly from a pooled `Dataset` instance should be retained after returning the `Dataset` instance to the pool.

- DatasetReader and DatasetWriterReader

  The `DatasetReader` object, obtained using the `Dataset.reader()` method, and the `DatasetWriterReader` object, obtained using the `Dataset.writerReader()` method, are thread-safe objects providing read-only and read/write access to the Dataset instance from which each was obtained. While one might consider pooling these objects, obtaining an instance is an inexpensive operation - the added complication of pooling instances of these objects would not be worth the trouble. In addition, operations against a `DatasetReader` or `DatasetWriterReader` should be completed before returning a `Dataset` instance to the pool.

- RecordStream and MutableRecordStream

  The `RecordStream` and `MutableRecordStream` objects, obtained from the `DatasetReader.records()` and `DatasetWriterReader`.records() methods, respectively, are the roots of the dataset bulk processing API based on Java streams. As with the `DatasetReader` or `DatasetWriterReader` instance from which it was obtained, a `RecordStream` or `MutableRecordStream` instance should not be retained or operated upon beyond the return of the pooled `Dataset` through which the stream instance was obtained. To ensure proper operation, stream instances must be closed before the pooled `Dataset` is returned. Additionally, `Iterator` and `Spliterator` instances obtained from a `RecordStream` or `MutableRecordStream` must not be retained or operated upon

after returning the pooled `Dataset`. If you want to use stream results as input to other work units, you must either drain the stream into a local data structure (which is then used to feed other work units) or use a non-pooled `DatasetManager` instance and `Dataset` instance having a lifecycle compatible with the lifetime of the stream.

- ReadRecordAccessor and ReadWriteRecordAccessor

  The `ReadRecordAccessor` and `ReadWriteRecordAccessor` extend key-based operations with conditional execution and CAS capabilities. These are obtained using the `on(…)` methods of the `DatasetReader` and `DatsetWriterReader` objects. Like the other objects in this group, operations performed using a `ReadRecordAccessor` or a `ReadWriteRecordAccess` must be complete before returning the `Dataset` instance from which they were obtained to the pool.

- AsyncDatasetReader and AsyncDatasetWriterReader

  Obtained using the `async()` methods of a `DatasetReader` or `DatasetWriterReader` instance, the `AsyncDatasetReader` and `AsyncDatasetWriterReader` objects provide non-blocking access to the TCStore API. In general, the methods on each return an `Operation` instance, implementing both the `java.util.concurrent.CompletionStage` and `java.util.concurrent.Future` interfaces, providing a full range of asynchronous task completion options. As with the objects discussed above, these operations should be completed before returning a `Dataset` instance to the pool.

Pooling Strategies

When seeking to improve application performance through resource pooling, the general recommendations are to:

1. Obtain one or more `ClusteredDatasetManager` instances.
   a. Pre-obtain during application initialization or defer until needed as appropriate for the application.
   b. Use no more than the number of `ClusteredDatasetManager` instances required to handle the application load.
   c. Do not "pool" the `ClusteredDatasetManager` instances in the traditional way - most applications do not need access to a `DatasetManager` instance so there's no need to "share out" a `ClusteredDatasetManager` instance. Instead, the `ClusteredDatasetManager` instances are used internally by the pooling implementation to support obtaining `Dataset` instances.
   d. Track `Dataset` instances obtained from each `ClusteredDatasetManager` instance; when no `Dataset instance` obtained from a `ClusteredDatasetManger` remains open, the `ClusteredDatasetManager` instance is idle and may be closed. Keeping around idle `ClusteredDatasetManger` instances for a certain amount of time may be appropriate for the application.
2. Pool `Dataset` instances for sharing - `Dataset` references can and should be shared.
   a. Use a strategy appropriate for your application to either pre-obtain a core set of `Dataset` instances during application initialization or defer allocation until demanded.
   b. Obtain no more than one (1) `Dataset` instance for a given dataset (name/type) per `ClusteredDatasetManager`.

c. Share `Dataset` instances by reference count. If appropriate for the application, a `Dataset` instance having no uses and left idle for some period of time should be closed.

d. Pair each `Dataset` instance with the `ClusteredDatasetManager` through which it was allocated.

Note:
If a `StoreReconnectFailedException` is raised for a TCStore operation, the `ClusteredDatasetManager` instance from which the object on which that operation was performed is disabled and must be discarded along with any `Dataset` instances obtained from that `ClusteredDatasetManager`. Once an object obtained from a `ClusteredDatasetManager` instance throws a `StoreReconnectFailedException`, all subsequent operations for that `ClusteredDatasetManager` instance will also throw a `StoreReconnectFailedException`. For pool management, the failing `ClusteredDatasetManager` instance must be replaced with a new instance. See "Clustered Reconnection" on page 21 for details.

3 Textual Query Language Extension

Reference

Concepts

Textual Querying of TCStore

TCStore API provides a native streaming API for analyzing Dataset contents. As an extension to that API, the Textual Query Language API (abbreviated TQL API) provides an interface for an ad-hoc querying of a dataset using textual queries. During runtime of the application, the dataset contents can be analyzed with queries whose logic is dynamic and not yet known during design time of the application.

The corresponding textual queries are based on SQL-like semantics. Using a query string, the user can describe in a declarative way how the data in the dataset is to be analyzed without specifying the actual processing steps. TQL API is designed for reading and analyzing the contents of a dataset. Thus, it solely supports read-only SQL queries; other common SQL commands for data creation or manipulation are not supported. Corresponding logic has to be defined using the native API.

Structured Data Access

The data model of TCStore is schema-less by design, i.e., the records in a dataset do not have to follow a common schema. Records in a dataset can share the same set of cells, but can also be based on completely different sets of cells. As a consequence a dataset can have heterogeneous contents.

By contrast, the TQL query approach requires a structured view on the data. Therefore, prerequisite for running a TQL query is to define a fixed schema, i.e., a subset of the data with a fixed structure. During query processing, each record of the dataset is projected to the set of cells constituting that schema.

TQL API offers the required operations to specify such a fixed schema, define a query, and consume the results.

Operations

Full Example

With TQL API users can query a dataset, more precisely the records of a dataset and their corresponding cells as well as their keys.

The following code sets up a TQL environment and executes a simple query:

```
final DatasetReader<String> reader = dataset.reader();   // 1
TqlEnvironment env = new TqlEnvironment(reader, // 2
        "Comedians",  // 3
        CellDefinition.defineString("FirstName"),  // 4
        CellDefinition.defineString("LastName"));
```

```
try (ResultStream resultStream =
        env.query("SELECT * FROM Comedians").stream()) {   // 5
    resultStream.forEach(System.out::println);  // 6
}
```

1	A `DatasetReader` is required to read the contents from a `Dataset`.
2	A `TqlEnvironment` is created, which takes as parameter the `DatasetReader`,
3	A `String` alias for referencing the `Dataset` in a query,
4	and var-args of `CellDefinition`, which defines the input schema used for processing the `Record` instances of the `Dataset`. Another variant uses a `Set` of cell definitions.
5	The `query` method is used to submit a TQL query string. Within that query string the alias of the `TqlEnvironment` instance is used to refer to the `Dataset`. The query takes all cell definitions into account that are defined in the input schema. If the query is valid, the `query` method returns a `Result` instance. The `Result` instance provides a `stream` method which delivers a `ResultStream` instance.
6	As `ResultStream` extends the `java.util.stream.Stream` interface, the results can be consumed in a streaming fashion, in this example by printing all results as terminal operation. The results themselves are instances of `CellCollection`, which is a collection of `Cell` instances.

Setup of Environment

As illustrated in the previous example, a `TqlEnvironment` instance is based on a `DatasetReader`, an alias, and a set of `CellDefinition` instances. Note that `TqlEnvironment` does not have an explicit lifecycle; it depends on the lifecycle of the `Dataset` and its associated `DatasetManager`.

Read Access

In order to run queries, the `DatasetReader` must be pointing to a `Dataset` that has not been closed. If the `Dataset` or the associated `DatasetManager` have been closed, query execution will fail with an exception.

Alias for Dataset

The mandatory alias for the `Dataset` defines the name under which the `Dataset` is referenced in a query. The alias is a non-empty string. If the alias consists of invalid characters, e.g. "name with space" or "?Id", an exception will be thrown.

Input Schema

A `TqlEnvironment` either uses an array or a set of `CellDefinition` instances to define the schema, i.e. the structured view on the data, which is used for querying a `Dataset`.

Consumption of Records

A query can only refer to the cell definitions of that schema, i.e., cells whose definitions have not been included in the schema, cannot be queried. Each `Record` of a `Dataset` is processed with respect to the defined input schema. For each cell definition of the input schema, it is checked whether

the record contains a corresponding cell. If so, the cell value is used in upstream query processing. If not, NULL is used. For example, let the input schema be "Name" of type `String`, "Age" of type `Integer`, "Weight" of type `Double`. Then the dataset entry [(Name,String,"Moe"), (Age,Integer,42)] will be internally processed as ["Moe",42,NULL]. NULL is also used if the cell definition name in the input schema equals the cell definition name of an existing cell, but only case-insensitively. This is due to names of cell definitions being handled case-sensitively.

Constraints on Cell Definitions

The cell definitions of the input schema must be unique with respect to the names. A reserved name is "key", which is used for accessing the key of a record. If two or more definitions share the same name, the input schema is rejected as TQL requires unique columns. It is also rejected if the names are equal case-insensitively, .e.g. "Age" and "age".

To deal with such ambiguous names, the API offers a manual and an automatic resolution approach.

Automatic Resolution of Ambiguities

The automatic resolution approach resolves ambiguities by introducing aliases for conflicting definitions. The method `resolveAmbiguousNames` can be applied to an array or a set of cell definitions, returning a set of cell definitions with new alias names. The alias name itself is built by appending "_" and the type in uppercase to the name. If the automatic resolution step introduces new ambiguities, an exception is thrown and the ambiguities have to be resolved manually.

The following example illustrates this approach:

```
CellDefinition<Integer> ageInt = CellDefinition.defineInt("Age"); // 1
CellDefinition<String> ageString = CellDefinition.defineString("Age");
TqlEnvironment env = new TqlEnvironment(reader,
        "Comedians",
        TqlEnvironment.resolveAmbiguousNames(ageInt, ageString));  // 2
```

1	The cell definitions are ambiguous as they have the same name.
2	The method `resolveAmbiguousNames` automatically resolves ambiguities by appending the type to the name of ambiguous cell definitions. In this example the generated aliases would be "Age_INT" and "Age_STRING".

Manual Resolution of Ambiguities

The manual approach is to introduce an alias for a cell definition with the alias name being unique. The method `as` maps an existing cell definition into a new cell definition with an alias.

The following example illustrates this approach:

```
CellDefinition<Integer> ageInt = CellDefinition.defineInt("Age");  // 1
CellDefinition<String> ageString = CellDefinition.defineString("Age");
TqlEnvironment env = new TqlEnvironment(reader,
        "Comedians",
        TqlEnvironment.as("Age_Resolved", ageString), ageInt);  // 2
```

1	The cell definitions are ambiguous as they have the same name.

2	The method as introduces an alias for a `CellDefinition`. The alias name must be unique. In this example, the alias is named "Age_Resolved", which is no more in conflict with the other cell definition named "Age".

The aliases, either introduced by the manual or the automatic approach, are then used for further query processing.

Sampling of Cell Definitions

When setting up a `TqlEnvironment`, the contents of the dataset and its structure may be unknown. In order to get an impression of the data, sampling can be used. More precisely, drawing a sample of records from the dataset and investigating their cell definitions provides a reasonable starting point for understanding the structure of the data. `TqlEnvironment` provides for that purpose the method `sampleCellDefinitions`. This method takes as parameters a `DatasetReader` and the sample size, the latter being greater than or equal to zero. If zero, the complete `Dataset` will be used as sample. Given such a sample, the superset of all cell definitions of records in the sample is determined and returned as a set.

Note:
It is important to note that sampling should only be used to get a first understanding of the dataset structure. As the contents of the dataset may change dynamically, so the sample of cell definitions may change. Also the set of cell definitions returned from sampling may contain ambiguities, i.e., definitions with the same name but a different type, or definitions whose names are case-insensitively equal. As a consequence the sample should not be used directly as the input schema, but should be investigated beforehand.

Inclusion of Record Key

Each record of a dataset consists of a key and a set of cells. By default, querying the key is not supported. To query the key, the method `TqlEnvironment.includeKeys` has to be called on a `TqlEnvironment` instance. Then the key can be used like every other cell definition of the input schema; the name of the corresponding column is "key".

The following example shows how to include the record key in query processing:

```
TqlEnvironment env = new TqlEnvironment(reader,
        "Comedians",
        CellDefinition.defineString("FirstName"),
        CellDefinition.defineString("LastName"))
        .includeKeys();  // 1
Result result = env.query("SELECT key FROM Comedians");  // 2
```

1	Given a `TqlEnvironment` instance, the key can be included by calling `includeKeys` in a fluent style.
2	Now a query can access the key of a record, using the "key" column.

Note:
If efficient key access is required for an application, the key should be additionally included in the value part of the record with an appropriate indexing setup.

Once the `TqlEnvironment` has been set up completely, it can be used to run an arbitrary number of queries. As the `TqlEnviromment` does not maintain any mutable state, it can therefore be used concurrently.

Querying of a Dataset

The method `query` of a `TqlEnvironment` instance takes as input the query string, which is based on the SQL-like semantics of TQL API. Within that query, all cell definitions of the input schema can be accessed, analogously to columns in a table being accessed in a SQL query. Note that within the TQL query, all identifiers for columns, functions, etc. are handled case-insensitively.

TQL API is targeted for reading from a `Dataset` and analyzing its contents. Therefore read-only SQL operations are allowed while data creation and manipulation operations like CREATE TABLE or UPDATE are not allowed. The query interface offers common SQL operations like filtering, projection, aggregation, or grouping. A query can only operate on one dataset; it cannot operate on multiple datasets using joins or other n-ary operations.

Each query has an output schema, describing what the query results look like. The output schema has to be compliant with TCStore type system. Otherwise the `query` method will fail with an exception. For example, a query like `SELECT CAST(Age AS BIGDECIMAL) AS FailType FROM Comedians` will fail. In such a case the query has to be adapted so that the output schema only contains types being available in TCStore type system. When calling the method `query` with a valid query string, it returns an instance of type `Result`. This instance is used for consuming the results and accessing the output schema.

Consumption of results

A `Result` instance provides access to the results of the query as well as to the schema of these results.

Streaming of Query Results

The actual consumption of the results is provided by the method `stream`. This method returns an instance of type `ResultStream`. Calling method `stream` multiple times will return `ResultStream` instances being executed independent of each other. As `ResultStream` extends `java.util.stream.Stream`, the results can be consumed in a streaming fashion using arbitrary follow-up intermediate or terminal operations. The results themselves are each a collection of cells. It is worth mentioning that the underlying query processing is done lazily, i.e., the results are not pre-computed in advance, but computed on demand when the next result is to be consumed. Each `ResultStream` instance can only be consumed once, as with `java.util.stream.Stream`. Note that the stream has to be closed in order to free resources. When using a terminal operation other than `iterator` or `spliterator`, it is automatically closed. However, it is good practice to close the stream explicitly.

The following example uses a try-with-resources statement to automatically close the stream after consuming all results.

```
try (ResultStream stream =
    env.query("SELECT * FROM Comedians").stream()) {
      stream.forEach(System.out::println);
}
```

Output Schema of Query Results

The result schema can be obtained by calling on a `Result` instance the method `cellDefinitions`, which returns `Collection<CellDefinition<?>>`. The cell definitions of the output schema can be used to retrieve the corresponding cell values from a query result entry. When composing the query results, NULL values are translated to absent cells in the resulting cell collection. For example, the output schema includes cell definition "LastName", but due to its result value being NULL, the resulting cell collection does not include that cell.

Note:
The alias-based resolution of ambiguities in the input schema may also affect the output schema. For example, resolving ambiguities automatically for input schema with `CellDefinition<Integer>` named "Age" and `CellDefinition<Long>` named "Age" will result for query "SELECT * FROM Source" in output schema `CellDefinition<Integer>` named "Age_INT" and `CellDefinition<Long>` named "Age_LONG".

Example

The following example illustrates the interplay of query schema and query results by plotting a table, using the result schema as header and the results as subsequent rows.

```
try(ResultStream resultStream =
    env.query("SELECT * FROM Comedians").stream()) {
    String header = resultSchema.stream()  // 1
            .map(cd -> String.format(" %10s ", cd.name()))
            .collect(Collectors.joining("|"));
    String rows = resultStream  // 2
            .map(row ->
                    resultSchema.stream()
                            .map(row::get)
                            .map(o -> String.format(" %10s ", o.orElse(null)))
                            .collect(Collectors.joining("|"))
            )
            .collect(Collectors.joining("\n")
            );
    System.out.println(String.format("%s%n%s", header, rows));  // 3
```

1	Using stream operations, the header is built using the `CellDefinition` instances of the result schema.
2	Using stream operations, the query results are consumed and converted into a `rows` string. Note that the cell collections are indexed by the cell definitions of the output schema. As mentioned earlier, if the value of a result column is NULL, the cell collection delivered as the result does not include such a cell. When using the corresponding cell definition to get that cell from the cell collection, an `Optional.empty()` object is returned.
3	Finally, header and rows are printed.

Insights into Query Execution

Each TQL query is parsed and translated into a physical query execution plan. That plan describes how logical operators are represented as physical operators, which indexes are used, and whether

all the data needs to be scanned. In case of performance bottlenecks of a query, its query execution plan can be examined to determine which physical operators are used or where indexes might speed up the execution.

For that reason, `ResultStream` also provides access to a representation of that query plan. The method `explainQuery` is an intermediate stream operation having a `java.util.stream.Consumer` as only parameter. That consumer consumes an `Object` instance, whose `toString` method returns a textual representation of the query execution plan. The object as well as content and form of the string representation are subject to change without notice. The plan is only available once the `ResultStream` has been closed, either by calling method `close` explicitly or using a terminal operation (other than `iterator` or `spliterator`).

Note:
The method `explainQuery` can only be called on a `ResultStream` instance. Adding subsequent stream operations to a `ResultStream` and calling then `explainQuery` is not supported.

For more details on the query execution plan and its string-based representation, see section "Performance Considerations" on page 57.

Usage and Best Practice

Application Scenarios

The TQL API can be used by developers or administrators to explore the contents of a dataset interactively. Such an interactive exploration can be useful for rapid prototyping, where the application logic often evolves dynamically. It is also useful for tools and applications which offer ad-hoc query functionality.

Using TQL as an abstraction layer, dynamic business logic can be developed, tested and deployed without the need to restart the server.

Interplay with Native Stream API

TQL API and native stream API complement each other, offering the user different options to develop business logic for an application. While the native API offers the complete range of CRUD functionality being specified in an imperative manner, TQL API offers ad-hoc query functionality being specified in a declarative manner. TQL API requires you to specify a schema before reading the data; the follow-up application then knows which data to expect. Native API does not require you to specify a schema upfront; it is up to the application to process the data in a structured manner.

The primary query interface for TCStore is the native API. Once business logic is designed and ready to be deployed into production, it should be specified using the native stream API as it offers the best performance.

Performance Considerations

Push-down of Query Logic

In order to understand potential performance implications when using TQL API, it is worth understanding the mechanisms running in the background. TQL API incorporates an internal query processing engine. When a TQL query is submitted, an optimization process decomposes the query into fragments. For each fragment it is checked whether it can be expressed in terms of the native stream API. Those fragments are then delegated to the TCStore client, which in turn tries to delegate them to TCStore server. In order to maximize server-side execution, the process of rewriting TQL query fragments in terms of native API tries to leverage portable functions of TCStore DSL whenever possible. The fragments which cannot be expressed in the native API are executed by the internal engine.

As the query processing engine is a logical part of the TCStore client, executing query fragments by the internal engine might require transferring a lot of data from TCStore server to client. Therefore, as a rule of thumb the queries should be written so that as much functionality as possible can be delegated to TCStore client.

The query execution plan can be utilized as tool for that purpose. By investigating the plan, the user can determine:

1. which fragments of the query cannot be expressed in TCStore's query DSL and therefore are executed by the internal engine on the client side;
2. which fragments of the query can be expressed in TCStore's query DSL; the resulting pipeline of stream operations is analyzed and decomposed into a sequence of portable operations and a sequence of non-portable operations;
3. which portable operations are evaluated remotely on the server and how the execution plan of the server is defined.

For details on portable and non-portable operations see also section “Stream Optimizations” on page 38.

Example

Let us examine a concrete query example and its execution plan.

```
Result result = env.query("SELECT * FROM Comedians WHERE Age**2 > 4"); // 1
try(ResultStream resultStream = result.stream()) {
    resultStream.explainQuery(System.out::println)  // 2
            .forEach(System.out::println);
}
```

1	The query filters all records where the squared age is greater than 4.
2	The corresponding query plan is printed.

Query Plan - No Push-Down

```
======= Query Plan Start =======
==== SQL Query:  // 1
SELECT * FROM Comedians WHERE Age**2 > 4
### WARN: Pushdown failed due to the following reasons:
###       WEPRME1953 - The expression of type EXPONENTIATION cannot be pushed down:
###       Age^CONSTANT_INTEGER.
==== Query operations evaluated by SQL engine:  // 2
SELECTION(filter[Age^CONSTANT_INTEGER>CONSTANT_INTEGER])
==== Query operations evaluated by TCStore Stream Query API: // 3
SOURCE(Comedians)
==== Code generated for TCStore Stream Query API:  // 4
IntCellDefinition Age_INT = CellDefinition.defineInt("Age");
StringCellDefinition FirstName_STRING = CellDefinition.defineString("FirstName");
IntCellDefinition Weight_INT = CellDefinition.defineInt("Weight");
StringCellDefinition LastName_STRING = CellDefinition.defineString("LastName");;
reader.records()
==== TCStore query plan: // 5
Stream Plan
    Structure:
        Portable:
            None
        Non-Portable:
           PipelineOperation{MAP(de.rtm.adapters.tcstore.source.translation.operators.
      OperatorTranslationResult$RecordStreamResult$$Lambda$470/28956604@1b5975f)}
            PipelineOperation{ITERATOR()}
    Server Plan: [stream id: a6e5eb73-47d7-4748-b260-53759d22471b]
        Stream Planning Time (Nanoseconds): 4290237
        Sorted Index Used In Filter: false
        Filter Expression: true
        Unknown Filter Count: 0
        Unused Filter Count And Filters (If Any): 0
        Selected Plan: Full Dataset Scan
======= Query Plan End =======
```

The query plan provides the following information:

1	The query is available under section "SQL Query". The warning indicates which fragments cannot be pushed down, in this example the exponentiation operation.
2	The query operations being executed by the internal query engine are listed. In this example this list comprises a selection using a filter on column age.
3	The query operations being executed by the native API only comprises the source itself, i.e. the access to the Dataset.
4	The optimization process rewrites query fragments in the native API. The corresponding code being generated is listed. In this example, only the method 'records' is called, delivering a stream of all instances in the dataset.
5	Finally the query plan of TCStore is listed. For details on that plan see also section “Stream Optimizations” on page 38 .

The example query can be rewritten, delivering the same results (under the assumption that age is positive).

```
result = env.query("SELECT * FROM Comedians WHERE Age > 2");  // 1
try(ResultStream resultStream = result.stream()) {
    resultStream.explainQuery(System.out::println)
            .forEach(System.out::println);
}
```

1	The filter predicate of the query has been rewritten.

The according query plan now illustrates the effects of rewriting the query on the push-down capabilities.

Query Plan - Push-Down

```
======= Query Plan Start =======
==== SQL Query:  // 1
SELECT * FROM Comedians WHERE Age > 2
==== Query operations evaluated by TCStore Stream Query API:   // 2
SELECTION(filter[Age>CONSTANT_INTEGER])
  SOURCE(Comedians)
==== Code generated for TCStore Stream Query API:  // 3
IntCellDefinition Age_INT = CellDefinition.defineInt("Age");
StringCellDefinition FirstName_STRING = CellDefinition.defineString("FirstName");
IntCellDefinition Weight_INT = CellDefinition.defineInt("Weight");
StringCellDefinition LastName_STRING = CellDefinition.defineString("LastName");;
reader.records()
.filter(Age_INT.exists()
   .and(Age_INT.intValueOr(0)
      .boxed()
      .isGreaterThan(2)))
==== TCStore query plan:  // 4
Stream Plan
    Structure:
        Portable:
            PipelineOperation{FILTER((CellDefinition[name='Age' type='Type<Integer>']
                .exists()&&(Age.intValueOr(0)>2)))}
        Non-Portable:
           PipelineOperation{MAP(de.rtm.adapters.tcstore.source.translation.operators

.OperatorTranslationResult$RecordStreamResult$$Lambda$470/28956604@fe32c2)}
             PipelineOperation{ITERATOR()}
    Server Plan: [stream id: a6e5eb74-47d7-4748-b260-53759d22471b]
        Stream Planning Time (Nanoseconds): 10704854
        Sorted Index Used In Filter: false
        Filter Expression: (CellDefinition[name='Age' type='Type<Integer>']
            .exists()&&(Age.valueOr(0)>2))
        Unknown Filter Count: 0
        Unused Filter Count And Filters (If Any): 0
        Selected Plan: Full Dataset Scan
======= Query Plan End =======
```

1	The query has been rewritten, no longer using the square operation.
2	Now the native API not only includes the source connection, but also the filter operation.
3	The generated code shows how the filter predicate is implemented.

4	The TCStore query plan also includes now the filter operation.

4 Transactions Extension

Overview

The Terracotta Store (TC Store) API provides operations with atomicity guarantees at a single `Record` level only. The Transactions Extension adds the ability to create transactions that involve multiple records belonging to one or more datasets. This feature is available in both the standalone and the clustered environments.

Note:
Only read committed transaction isolation is supported.

Transaction Controller

Transactions are managed through a `TransactionController` instance. There are two `createTransactionController` factory methods to create instances:

```
TransactionController transactionController =
  TransactionController.createTransactionController // 1
  (datasetManager,                                  // 2
    datasetConfigurationBuilder);                   // 3
```

1	A `TransactionController` instance is required to create and execute transactions.
2	A `DatasetManager` is needed to host the internal transaction dataset.
3	A `DatasetConfigurationBuilder` is used to configure this internal transaction dataset if it does not already exist.

If the transaction controller has already been created (and the internal transaction dataset already exists):

```
TransactionController controller1 =
TransactionController.createTransactionController(datasetManager);  // 1
```

1	A `DatasetManager` instance is required to load the internal transaction dataset that stores the existing metadata.

If the internal transaction dataset does not exist and the second form is used then a `DatasetMissingException` will be thrown.

Once retrieved, a `TransactionController` instance can be used to execute actions transactionally using one of the `TransactionController.execute(…)` methods or a transactional execution context can be built using the `TransactionController.transact()` method.

Note:
If a transaction is executed against a **persistent** dataset, then the internal transaction dataset should also be configured as **persistent** since transaction metadata is stored in both the datasets involved in the transaction and the internal transaction dataset. Failure to persist either set of metadata may leave transactions in an unexpected state after a restart.

Warning:
The contents of the internal transaction dataset must not be accessed or modified by the user.

Transaction timeouts

Every transaction has a timeout defined. If a transaction does not finish before it times out, it will get rolled back by the system. Any interaction with a transaction can throw an unchecked `StoreTransactionTimeOutException` if the transaction has timed out.

A `TransactionController` has a default transaction timeout defined as 15 seconds. This can be overridden at runtime by creating a derived instance with a new timeout:

```
TransactionController controller2 = transactionController.withDefaultTimeOut(50,
   TimeUnit.SECONDS);   // 1
```

1	A `TransactionController` created using a custom transaction timeout.

Timeouts can also be modified on a per transaction basis using a Transaction ExecutionBuilder. See the section "Transaction ExecutionBuilder" on page 64 for details.

Note:
The classes in the TCStore transactions framework predominantly follow the immutable builder design pattern. Simply calling the `withDefaultTimeout` method as shown above is ineffective if the `TransactionController` instance it returns is not captured/used.

Transaction Execution

The simplest way to execute a transaction is using one of the `TransactionController.execute` methods. The parameters to these methods define the transaction workload and the resources that take part in the transaction.

```
Long numberOfEmployees =
    transactionController.execute(employeeReader,                    // 1
    reader -> reader.records().count());                             // 2
Long numberOfEmployeesAndCustomers =
   transactionController.execute(employeeReader, customerReader,     // 3
  (empReader, custReader) ->
      empReader.records().count() + custReader.records().count());   // 4
transactionController.execute(employeeWriterReader,                  // 5
  (TransactionalTask<DatasetWriterReader<Integer>>) writerReader ->
      writerReader.on(1).delete());                                  // 6
```

1	`TransactionController.execute` here takes a `DatasetReader` as the resource that takes part in the transaction.
2	The `TransactionalAction` instance is conveniently expressed here as a lambda expression. The action is executed atomically with respect to the enrolled resources and its return value is then returned from `TransactionController`.execute(…).
3	Here two `DatasetReader` instances are passed in as resources.

4	The `TransactionalBiAction` defines a transaction over the two passed in resources. And the result is returned by the `TransactionController.execute(…)`.
5	Additional overloads of `TransactionController.execute(…)` take `DatasetWriterReader` instances as resources.
6	A `TransactionalTask` has no return value and is therefore only available for execution against writer-reader resources.

There are overloads of `TransactionController.execute(…)` defined to handle all one and two dataset transactions. More complicated transactions require a transaction context to be built.

Transaction ExecutionBuilder

When assembling a transaction that cannot be handled using one of the `TransactionController.execute` methods, a transaction `ExecutionBuilder` instance can be used. An `ExecutionBuilder` supports setting a transaction-specific timeout and identifying more `DatasetReader`/`DatasetWriterReader` resources to include in the transaction than is possible using the execute methods.

`Dataset` instances involved in a transaction must be identified to the transaction context by providing `DatasetReader` and/or `DatasetWriterReader` instances to the context through the `ExecutionBuilder.using` methods. Accesses to *undeclared* `Dataset` instances which are not under the scope of the transaction can result in unexpected behavior.

ReadOnlyExecutionBuilder

```
ReadOnlyExecutionBuilder
   readOnlyExecutionBuilder = controller1.transact()  // 1
  .timeout(100, TimeUnit.SECONDS)                     // 2
  .using("empReader", employeeReader)                 // 3
  .using("custReader", customerReader);               // 4
```

1	`transact()` in `TransactionController` returns a `ReadOnlyExecutionBuilder`, the starting point for all transaction builders, which is then used to construct a read-only transaction.
2	A transaction can have its own timeout defined through the `ExecutionBuilder`.
3	A `DatasetReader` is added as a resource to the execution builder. This resource will be used by the executing transaction.
4	Multiple resources can be added to an execution builder.

Note:
For a transaction in which no mutations will take place, using `DatasetReader` instead of `DatasetWriterReader` instances will result in a `ReadOnlyTransaction` which can provide improved performance.

ReadWriteExecutionBuilder

```
ReadWriteExecutionBuilder readWriteExecutionBuilder = controller2.transact()
  .using("custWriterReader", customerWriterReader)   // 1
  .using("empReader", employeeReader);               // 2
```

1	If a `DatasetWriterReader` is added as a resource to a `ReadOnlyExecutionBuilder`, then it returns a `ReadWriteExecutionBuilder` which can be used to execute a read/write transaction.
2	A `DatasetReader` can also be added to a `ReadWriteExecutionBuilder` as resource that could eventually get used by the read/write transaction executed using the `ReadWriteExecutionBuilder`.

Executing Using an ExecutionBuilder

The transactions API provides two ways of executing a transaction using an execution builder as described below.

Executing a TransactionalAction

One way to execute a transaction is to write a function that contains all the logic of the transaction and then execute that function using the execution builder.

```
Double totalSalary = transactionController.transact()
  .using("empReader", employeeReader)
  .execute(readers -> {                                    // 1
    DatasetReader<Integer> empTransactionalReader =
       (DatasetReader<Integer>) readers.get("empReader");  // 2
    double first100EmployeesSalaryCount = 0;
    for (int i = 1; i < 100; i++) {
      first100EmployeesSalaryCount +=
        empTransactionalReader.get(i).map(rec ->
            rec.get(SALARY).orElse(0D)).orElse(0D);        // 3
    }
    return first100EmployeesSalaryCount;                   // 4
  });
```

1	`ReadOnlyExecutionBuilder.execute()` takes a `TransactionalAction` instance as parameter. `TransactionalAction` is a functional interface with a method `perform()` taking in a single argument (`Map<String, DatasetReader>`) and returning an object. In this example, the `TransactionalAction` instance is formed from a lambda expression. This `TransactionalAction` is executed as a single transaction satisfying all the ACID properties.
2	The `Map` (`readers`) provides access to the `DatasetReader` instances that were added as resources.
3	Transactional read operations can be performed using these extracted dataset readers.
4	The value returned by `perform()` of `TransactionalAction` is returned by the `execute()` method here.

Note:
For proper transactional operations, use **only** `DatasetReader` instances obtained from the `Map` and not from variables defined outside of the scope of the lambda and captured within the lambda.

```
int numberOfRecordsUpdated = transactionController.transact()
  .using("empReader", employeeReader)
  .using("empWriterReader", employeeWriterReader)
  .execute((writerReaders, readers) -> {                          // 1
    DatasetWriterReader<Integer> empTransactionalWriterReader =
      (DatasetWriterReader<Integer>)
          writerReaders.get("empWriterReader");                   // 2
    DatasetReader<Integer> empTransactionalReader =
      (DatasetReader<Integer>) readers.get("empReader");          // 2
    int numRecordsUpdated = 0;
    for (int i = 1; i < 100; i++) {
      numRecordsUpdated += empTransactionalWriterReader.on(i)     // 3
        .update(UpdateOperation.write(SALARY).doubleResultOf(
            SALARY.doubleValueOr(0D).add(100D)))
        .isPresent() ? 1 : 0;
    }
    System.out.println("Total Employee = " +
        empTransactionalReader.records().count());                // 3
    return numRecordsUpdated;                                     // 4
  });
```

1	`ReadWriteExecutionBuilder.execute()` takes a `TransactionalBiActioninstance` as parameter. `TransactionalBiAction` is a functional interface with a method `perform()` taking two arguments (`Map<String, DatasetWriterReader>` and `Map<String, DatasetReader>`) and returning an object. In this example, the `TransactionalBiAction` instance is formed from a lambda expression. This `TransactionalBiAction` is executed as a single transaction satisfying all the ACID properties.
2	The maps (`writerReaders`, `readers`) provide access to the `DatasetWriterReader` and `DatasetReader` instances that were added as resources.
3	Transactional CRUD operations can be performed using these extracted dataset readers and writerReaders.
4	The value returned by `perform()` of `TransactionalBiAction` is returned by the `execute()` method here.

Note:
For proper transactional operations, use **only** `DatasetWriterReader` and `DatasetReader` instances obtained from the maps and not from variables defined outside of the scope of the lambda and captured within the lambda.

External Transaction Control

Another way of executing a transaction is by creating a `Transaction` instance using an execution builder. This `Transaction` instance is then used to extract the transactional versions of the dataset readers and writerReaders that were added as resources. These instances are then used to perform

all the transactional activity and finally the transaction is committed through the `Transaction` instance.

Note:
External transaction control is deprecated in favor of the execution using lambda forms described above.

```
ReadOnlyTransaction readOnlyTransaction =
  transactionController.transact()
  .using("empReader", employeeReader)
  .begin();                                                    // 1
boolean exceptionThrown1 = false;
try {
  DatasetReader<Integer> empTransactionalReader =
      readOnlyTransaction.reader("empReader");                 // 2
  double first100EmployeesSalarySum = 0;
  for (int i = 1; i <= 100; i++) {
    first100EmployeesSalarySum +=
      empTransactionalReader.get(i).map(rec ->
          rec.get(SALARY).orElse(0D)).orElse(0D);              // 3
  }
  System.out.println("First 100 employee's salary sum = " +
      first100EmployeesSalarySum);
} catch (Exception e) {
  exceptionThrown1 = true;
  e.printStackTrace();
  readOnlyTransaction.rollback();
}
if (exceptionThrown1 == false) {
  readOnlyTransaction.commit();                                // 4
}
```

1	`ReadOnlyExecutionBuilder.begin()` returns a `ReadOnlyTransaction` instance.
2	The `ReadOnlyTransaction` instance is then used to retrieve transactional dataset readers.
3	The readers can then be used for performing transactional read operations.
4	Finally, to finish the transaction `commit()` or `rollback()` is called on the `ReadOnlyTransaction` instance.

```
TransactionController.ReadWriteTransaction
  readWriteTransaction = transactionController.transact()
  .using("empReader", employeeReader)
  .using("empWriterReader", employeeWriterReader)
  .begin();                                                      // 1
boolean exceptionThrown = false;
int numRecordsUpdated = 0;
try {
  DatasetWriterReader<Integer> empTransactionalWriterReader =
    readWriteTransaction.writerReader("empWriterReader");       // 2
  DatasetReader<Integer> empTransactionalReader1 =
    readWriteTransaction.reader("empReader");                   // 2
  for (int i = 1; i < 100; i++) {
    numRecordsUpdated += empTransactionalWriterReader.on(i)     // 3
      .update(UpdateOperation.write(SALARY).doubleResultOf(
          SALARY.doubleValueOr(0D).add(100D)))
```

```
      .isPresent() ? 1 : 0;
  }
  System.out.println("Total Employee = " +
      empTransactionalReader1.records().count());              // 3
} catch (Exception e) {
  exceptionThrown = true;
  e.printStackTrace();
  readWriteTransaction.rollback();
}
if (exceptionThrown == false) {
  if (numRecordsUpdated == 100) {
    readWriteTransaction.commit();                             // 4
  } else {
    readWriteTransaction.rollback();                           // 4
  }
}
```

1	`ReadWriteExecutionBuilder.begin()` returns a `ReadWriteTransaction` instance.
2	The `ReadWriteTransaction` instance is then used to retrieve transactional dataset readers and writer-readers.
3	The readers and writer-readers can then be used for performing transactional operations.
4	Finally, to finish the transaction, `commit()` or `rollback()` is called on the `ReadWriteTransaction` instance. For read/write transactions the `commit()` or `rollback()` will resolve the dataset states to their correct forms.

Transactional Operation Behavior

Read

Transactional reads are non-blocking and are executed at read committed isolation level. A read operation will return the latest committed image of the record and will not be blocked by any active transaction.

Add, Delete and Update

Write operations on a record written by another active transaction will wait for that transaction to finish. This means that a record dirtied by a transaction cannot be updated by another transaction until the first transaction is committed, rolled-back or times out.

Stream Operations

Only non-mutative transactional record stream operations are supported. Reads performed by stream operations have the same semantics as the simple reads described above.

Best practices

1. A non-persistent internal transaction dataset should **only** be used if you are sure transactions will never involve records from a persistent `Dataset`. Using a non-persistent internal transaction dataset when appropriate can provide better performance however. For the most reliable behavior, datasets enrolled in transactions should use the same persistence mechanism as the `TransactionController` itself.
2. Read-only transactions should be used for transactions with no write operations to get better performance.
3. A `TransactionAction` or `TransactionBiAction` should only use the provided transaction resources. Transactional guarantees are provided only for the operations performed through the added resources.
4. Avoid executing transactions using External Transaction Control (see the description in the section "Executing Using an ExecutionBuilder" on page 65). This form of executing is significantly more error prone than the recommended functional forms.