TCStore API Developer Guide

Version 10.1

October 2017

TERRACOTTA

This document applies to TCStore API Version 10.1 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: STO-DG-101-20171103

Table of Contents

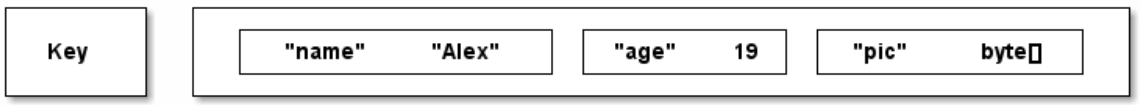
1 Reference

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.

Figure 1. 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>
- 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>
- 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. 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.

The Cell instances within a Record are unordered.

Read and Write Visibility

Read and write visibilities are the two settings that are required to read or write into a `Dataset`. The Read Visibility setting of a reader defines whether or not it will see the latest changes made by other writers into the `Dataset`. The Write Visibility setting of a writer defines how soon its own changes are made visible to other readers of the `Dataset`.

Read visibility

There are two kinds of read visibility:

- **Routine read visibility**: This provides standard visibility of a write by an other writer based on its write visibility setting.
- **Definitive read visibility**: This provides immediate visibility of the latest write by writers regardless of their write visibility setting.

Write visibility

There are two kinds of write visibility:

- **Eventual write visibility**: Visibility of these writes are eventual or immediate depending on the reader read visibility setting.
- **Immediate write visibility**: Visibility of these writes are immediate to all the readers, regardless of their reader settings.

Setting up read and write visibility

```
DatasetReader<Integer> routineReader =
    dataset.reader(ROUTINE.asReadSettings()); // <1>
DatasetReader<Integer> definitiveReader =
    dataset.reader(DEFINITIVE.asReadSettings()); // <2>
```

1	`Dataset.reader()` returns a `DatasetReader` that is used to read from the dataset. This method takes in `ReadSettings` as a parameter.
2	Every `DatasetReader` has a read visibility set for all the reads it performs. This read visibility setting is part of the `ReadSettings` passed into `Dataset.reader()` as parameter.

```
DatasetWriterReader<Integer> writerReader1 =
    dataset.writerReader(DEFINITIVE.asReadSettings(), EVENTUAL.asWriteSettings());  // <1>
DatasetWriterReader<Integer> writerReader2 =
    dataset.writerReader(DEFINITIVE.asReadSettings(), IMMEDIATE.asWriteSettings()); // <2>
```

1	`Dataset.writerReader()` returns a `DatasetWriterReader` that is used to read and write into the dataset. This method takes in `ReadSettings` and `WriteSettings` as parameters.

2	Every `DatasetWriterReader` has a read and write visibility set for all the reads and writes it performs. The read and write visibility are part of the `ReadSettings` and `WriteSettings` passed into `Dataset.writerReader()` as parameters.

Choosing a read and write visibility

The following table illustrates the behavior of the various pairs of read and write consistency settings.

	ROUTINE read visibility	DEFINITIVE read visibility
EVENTUAL write visibility	eventual	immediate
IMMEDIATE write visibility	best perf immediate	avoid (immediate)

If only eventual visibility is required then writes should be `EVENTUAL`, and reads should be `ROUTINE`. If only immediate visibility is required then writes should be `IMMEDIATE` and reads should be `ROUTINE`. For mixed consistency it is possible to use `EVENTUAL` for writes and let readers choose the consistency: `ROUTINE` for eventual, `DEFINITIVE` for immediate.

Configuration and Lifecycle Operations

Full example

The following code configures a client-side cache over a new clustered `Dataset`:

```
try (DatasetManager datasetManager = DatasetManager.clustered(clusterUri) // <1>
    .withCache(DatasetManager.cacheConfiguration()  // <2>
            .heap(256, MemoryUnit.MB)  // <3>
            .offheap(2, MemoryUnit.GB)) // <4>
    .build()) {  // <5>
  DatasetConfiguration ordersConfig = datasetManager.datasetConfiguration()  // <6>
      .offheap("offheap-resource-name")  // <7>
      .disk("disk-resource-name")  // <8>
      .build();  // <9>
  try (Dataset orders =
      datasetManager.createDataset("orders", Type.LONG, ordersConfig)) {  // <10>
  }
}
```

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 `withCache` method on the `DatasetManagerBuilder` configures the cluster client to use client side caching. A `CacheConfigurationBuilder` is passed to `withCache` which is constructed using the static method `DatasetManager.cacheConfiguration`.
3	The client side cache is configured to use 256MB of heap memory.
4	The client side cache is configured to use 2GB of offheap memory.
5	The configuration of the `DatasetManager` is now completed and an instance of `DatasetManager` is created. This instance represents a connection to the cluster. `DatasetManager` is `AutoCloseable` so try-with-resources should be used.
6	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.
7	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.
8	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.
9	The specification of the `DatasetConfiguration` is now completed and an instance is created.
10	A new `Dataset` called `orders` is created. It has a key of type `LONG`. `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.

Caching

Caches can be configured at the `DatasetManager` level, as shown in the full example above, in which case the cache is shared across any `Dataset` managed by that `DatasetManager`. Alternatively a cache can be configured for a specific `Dataset` using a variant of `getDataset`:

```
try (DatasetManager datasetManager = DatasetManager.clustered(clusterUri).build();
     Dataset orders = datasetManager.getDataset(
         "orders", Type.LONG, DatasetManager.cacheConfiguration()
         .heap(256, MemoryUnit.MB)
         .offheap(2, MemoryUnit.GB))) {
}
```

Caching is optional. If you want to use the TCStore API with no cache then do not call `withCache` and use the variant of `getDataset` that does not take a `CacheConfigurationBuilder`.

Cache tiers

The TCStore API has the concept of offheap storage where data is stored in memory that is outside the JVM's control. The advantage of this is that lots of data can be stored without impacting garbage collection. The memory is managed directly by TCStore. However, there is a trade-off: access times for data in offheap are longer because the data must be brought into the Java heap before it can be used. In practice, for larger caches, configuring offheap storage improves performance.

A cache need not have any offheap storage configured, in which case all cached records will be held in the Java heap. This is more suitable for smaller caches.

If a cache is configured with both `heap` and `offheap` as in the full example above, the Java heap holds the hottest records, less-hot records will be held in offheap.

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 no disk resource is specified, then data is held just in the memory of the servers of the cluster.

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.withCache(cacheConfiguration);
DatasetManager datasetManager = builder.build();
```

will create a clustered `DatasetManager` that has no client side cache because the build is called on the wrong object.

Instead use the following form:

```
ClusteredDatasetManagerBuilder builder = DatasetManager.clustered(clusterUri);
ClusteredDatasetManagerBuilder cachedBuilder = builder.withCache(cacheConfiguration);
DatasetManager datasetManager = cachedBuilder.build();
```

or more fluently:

```
DatasetManager datasetManager = DatasetManager.clustered(clusterUri)
    .withCache(cacheConfiguration)
    .build();
```

Operations

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. Both can be obtained with required read and write visibility settings.

```
DatasetWriterReader<String> writerReader =
    persons.writerReader(ROUTINE.asReadSettings(),
    IMMEDIATE.asWriteSettings()); // <1>
```

1	The `Dataset.writerReader` method needs two arguments - `ReadSettings` and `WriteSettings`, which returns the required `DatasetWriterReader`. Similarly, the `Dataset.reader` method returns a `DatasetReader`.

Adding Records

```
// define some keys
String person1 = "p1";
```

```
String person2 = "p2";
String person3 = "p3";
writerReader.add(person1, // <2>
    Person.FIRST_NAME.newCell("Marcus"), // <3>
    Person.LAST_NAME.newCell("Aurelius"),
    Person.RATED.newCell(true),
    Person.NICENESS.newCell(0.65D));
```

2	The `DatasetWriterReader` API provides the `add` method which takes the specified key of the record,
3	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 `explainConsumer` 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.
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,
      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,
        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 15.

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);
Comparator<Record<?>> comparator = intValue.asComparator(); // <1>
ToIntFunction<Record<?>> incremented = intValue.increment(); // <2>
```

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

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.

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, 42); // <2>
UpdateOperation.CellUpdateOperation<Long, Integer>
   increment = UpdateOperation.write(defnA, defnA.intValueOr(0).increment()); // <3>
UpdateOperation.CellUpdateOperation<Long, Integer>
   copy = UpdateOperation.write(defnB, 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.

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

```
DatasetManager datasetManager = DatasetManager.clustered(clusterUri).build();
DatasetConfiguration configuration = datasetManager.datasetConfiguration()
  .offheap("offheap-resource-name")
  .index(CellDefinition.define("orderId", Type.STRING),
        IndexSettings.BTREE) // <1>
  .build();
Dataset<Long> dataset =
    datasetManager.createDataset("indexedOrders", Type.LONG, configuration);
Indexing indexing = dataset.getIndexing(); // <2>
Operation<Index<Integer>> indexOperation =
    indexing.createIndex(CellDefinition.define("invoieId", Type.INT),
        IndexSettings.BTREE); // <3>
Index<Integer> invoiceIdIndex = indexOperation.get(); // <4>
indexing.getAllIndexes(); // <5>
indexing.getLiveIndexes(); // <6>
indexing.destroyIndex(invoiceIdIndex); // <7>
```

Creating Secondary Indexes

1	An `Index` can be created while the dataset is being created. The `DatasetConfigurationBuilder#index` 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

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

Destroying Indexes

7	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 26 and "Index Use" on page 31. 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 26.

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 `explainConsumer` 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 15, `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 20). 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 29 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 31 below.

DSL Support for Portable Operations

As discussed in the section "Functional DSL" on page 20, 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.

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_CLASSCellDefinition`, 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 `filterPredicate` 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.