

TCStore API Developer Guide

Version 10.2

April 2018

This document applies to TCStore API Version 10.2 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.

Copyright © 2010-2018 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at <http://softwareag.com/licenses>.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at <http://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at <http://softwareag.com/licenses> and/or in the root installation directory of the licensed product(s).

Table of Contents

Reference.....	5
Concepts.....	6
Data Model.....	6
Configuration and Lifecycle Operations.....	8
Operations.....	10
Clustered Reconnection.....	10
CRUD Operations.....	10
Streams.....	12
Asynchronous API.....	17
Functional DSL.....	17
Indexes.....	20
Usage and Best Practices.....	23
Stream Optimizations.....	24
Textual Query Language Extension.....	31
Reference.....	32
Concepts.....	32
Operations.....	32
Usage and Best Practice.....	39
Application Scenarios.....	39
Interplay with Native Stream API.....	39
Performance Considerations.....	39

1 Reference

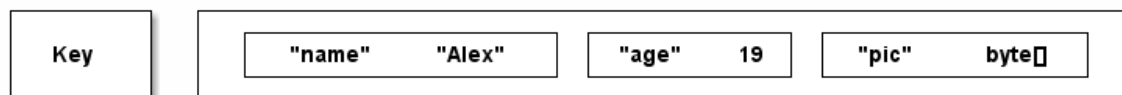
■ Concepts	6
■ Configuration and Lifecycle Operations	8
■ Operations	10
■ Functional DSL	17
■ Indexes	20

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.

Configuration and Lifecycle Operations

Full example

The following code configures a new clustered Dataset:

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

1	The static method <code>DatasetManager.clustered</code> starts the process of configuring a clustered <code>DatasetManager</code> . It returns a <code>DatasetManagerBuilder</code> which allows configuration of the cluster client.
2	The <code>DatasetManager</code> is created, represents a connection to the cluster. <code>DatasetManager</code> is <code>AutoCloseable</code> so <code>try-with-resources</code> should be used.
3	A <code>DatasetConfiguration</code> is required to create a new <code>Dataset</code> . A <code>DatasetConfigurationBuilder</code> that can be used to construct a <code>DatasetConfiguration</code> is acquired using the method <code>datasetConfiguration</code> on the <code>DatasetManager</code> . Note that a <code>DatasetConfiguration</code> should be used with the <code>DatasetManager</code> 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.
6	The specification of the <code>DatasetConfiguration</code> is now completed and an instance is created.
7	A new <code>Dataset</code> called <code>orders</code> is created. It has a key of type <code>LONG</code> .

8	The previously created dataset is retrieved. <code>Dataset</code> is <code>AutoCloseable</code> so <code>try-with-resources</code> 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 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.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();
```

Operations

Clustered Reconnection

Important: Operations against a clustered `Dataset`, `DatasetReader` or `DatasetWriterReader` (or any other derived instances) may throw a `StoreOperationAbandonedException` if a server connection is lost during the operation. In such circumstances inspection of the state upon reconnect will be required to determine the effect (if any) of the abandoned operation.

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	<code>Dataset.writerReader</code> returns the required <code>DatasetWriterReader</code> for mutative access. Similarly, <code>Dataset.reader</code> returns a <code>DatasetReader</code> .
---	--

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 <code>DatasetWriterReader</code> API provides the <code>add</code> method which takes the specified key of the record,
2	And var-args of <code>Cell</code> . Another variant takes an <code>Iterable</code> of cells.

Accessing Records

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

1	DatasetReader has a <code>get</code> method which takes a key as the argument. It returns an <code>Optional</code> of record. If the dataset doesn't have a record against the provided key <code>Optional.isPresent</code> 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 <code>DatasetWriterReader.update</code> method requires the key of the record to be updated along with an <code>UpdateOperation</code> of the cell. The <code>UpdateOperation.write</code> 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 <code>UpdateOperation.allOf</code> which takes a var-arg. To remove a cell use <code>UpdateOperation.remove</code> . 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	<code>DatasetWriterReader.delete</code> takes key and returns <code>true</code> 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 <code>DatasetWriterReader#on</code> which takes a key as the argument. <code>DatasetReader#on</code> returns a <code>ReadRecordAccessor</code> which has read only access to the record.
2	The <code>read</code> method takes a <code>Function</code> as an argument which maps the record to the required output.
3	The <code>upsert</code> method is like the same verb in a RDBMS: it will <code>add</code> if the record is absent or <code>update</code> if the record is present.
4	There is an advanced <code>update</code> that takes an additional <code>BiFunction</code> as <code>mapper</code> along with an <code>UpdateOperation</code> . 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. <code>ReadWriteRecordAccessor#iff</code> takes a predicate, the operations done on <code>ConditionalReadWriteRecordAccessor</code> are supplied with the same predicate. If the predicate returns <code>true</code> , the operation is executed against the record.
6	If the predicate returns <code>true</code> , the read will return a record.
7	The record will only be updated if the predicate returns <code>true</code> .
8	Similarly, the deletion succeeds if the predicate was <code>true</code> .

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 <code>DatasetReader.record()</code> method returns a <code>RecordStream</code> delivering <code>Record</code> instances from the <code>Dataset</code> referred to by the <code>DatasetReader</code> .
2	Stream intermediate operations on a <code>RecordStream</code> return a stream whose type is determined by the operation and its parameters. In this example, <code>filter</code> provides a <code>RecordStream</code> .
3	A Stream terminal operation on <code>RecordStream</code> produces a value or a side-effect. In this case, <code>count</code> returns the number of <code>Record</code> 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	<p>The <code>RecordStream.explain</code> operation observes the stream pipeline and provides the pre-execution analysis information for this stream pipeline. It takes, as a parameter, a <code>Consumer</code> which is passed an explanation of the stream execution plan. <code>RecordStream.explain</code> returns a <code>RecordStream</code> for further pipeline construction. For <code>explain</code> to be effective, the pipeline must be terminated - the plan is not determined until the pipeline begins execution. The <code>explainConsumer</code> is called once the pipeline is closed. For a <code>RecordStream</code> against a clustered <code>TCStore</code> configuration, the explanation identifies the operations in each of the server-side and client-side pipeline segments.</p>
2	<p>In a clustered configuration, a <code>RecordStream</code> <i>batches</i> 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 <code>RecordStream.batch</code> operation. The <code>batch</code> operation takes a batch size as parameter and uses it as a hint for the batch size to use when transferring elements. <code>RecordStream.batch</code> returns a <code>RecordStream</code> for further pipeline construction.</p>
3	<p>The <code>RecordStream.log</code> method produces a <code>Consumer</code> for use in <code>Stream.peek</code> to log a message according to the specified format and arguments. The first argument provides a message format like that used in the <code>SLF4J MessageFormatter.arrayFormat</code> 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 <code>SLF4J</code> (the logging abstraction used in <code>TCStore</code>). If the <code>peek(log (...))</code> operation is in the server-side pipeline segment, the formatted message is logged on the <code>TCStore</code> server. If the <code>peek(log (...))</code> operation is in the client-side segment, the formatted message is logged in the client.</p>
4	<p>The <code>RecordStream.inline</code> operation disables the element batching discussed above. When <code>inline</code> is used, each stream element is processed by both the server-side and client-side pipeline segments before the next element is processed. <code>RecordStream.inline</code> returns a <code>RecordStream</code> for further pipeline construction.</p>

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

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	<code>MutableRecordStream.deleteThen()</code> is an intermediate operation and deletes every <code>Record</code> in the stream.
---	---

2	<code>MutableRecordStream.deleteThen()</code> returns a <code>Stream</code> of the deleted <code>Record</code> instances. Note that it does not return a <code>RecordStream</code> or a <code>MutableRecordStream</code> .
---	--

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	<code>MutableRecordStream.mutate()</code> takes in <code>UpdateOperation</code> as parameter and performs the update transformation against the <code>Record</code> instances in the stream. This is a terminal operation and returns nothing.
---	--

delete

The `MutableRecordStream.delete` operation is a terminal operation that 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	<code>MutableRecordStream.delete()</code> deletes every <code>Record</code> 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 12.

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 <code>async()</code> method on an existing reader or writer-reader.
2	Methods then create asynchronous executions represented by <code>Operation</code> instances.
3	Operations can be composed with other operations using the usual <code>CompletionStage</code> methods.
4	Operations can also be composed with synchronous operations still yielding <code>Operation</code> instances.
5	<code>Operation</code> also extends <code>Future</code> 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 <code>Function<Record<?>, Optional<String>></code> 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.

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 <i>install</i> 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 <code>cell-a</code> from the resultant bi-function's two arguments.
2	Extract the output value of <code>cell-a</code> from the resultant bi-function's two arguments.
3	Extract the value of <code>cell-a</code> from the first value of the resultant function's tuple argument.
4	Extract the value of <code>cell-a</code> 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 = indexing.get(); // 4

```

1	An Index can be created while the dataset is being created. The <code>DatasetConfigurationBuilder.index</code> method takes a <code>CellDefinition</code> and an <code>IndexSettings</code> . Currently only <code>IndexSettings.BTREE</code> 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 <code>Dataset.getIndexing</code> to create/delete indexes on a dataset.
3	The <code>Indexing.createIndex</code> method again takes a <code>CellDefinition</code> and an <code>IndexSettings</code> , to return an <code>Operation of Index</code> . <code>Operation</code> 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	<code>Indexing.getAllIndexes</code> returns all the indexes created on the dataset, regardless of their status.
2	<code>Indexing.getLiveIndexes</code> 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 <code>Indexing.destroyIndex</code> .
---	---

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 24

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 24 and "[Index Use](#)" on page 29. There is also a tool, the *stream plan*, that provides visibility on the effectiveness of the performance methods; this is described in the following section "[Stream Plan](#)" on page 24.

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 12, `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 17). 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 27 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 <i>lambda expressions</i> (expressions using the "arrow" operator) always produce non-portable operations.
2	The map operation in this example <i>could</i> 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.valueOrNull()) // 2
    .collect(toList());
```

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

Stream Plan - Portable Operations:

```
Stream Plan
  Structure:
    Portable:
      PipelineOperation{FILTER((class==mammal))} // 1
      PipelineOperation{MAP(class.valueOrNull())}
    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 <code>toList</code> 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 <code>class</code> (<code>TAXONOMIC_CLASS</code>) cell was defined. See the section "Index Use" on page 29 below.

DSL Support for Portable Operations

As discussed in the section ["Functional DSL" on page 17](#), 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:

<code>averagingDouble</code>	<code>groupingBy</code>	<code>partitioningBy</code>
<code>averagingInt</code>	<code>groupingByConsumer</code>	<code>summingDouble</code>
<code>averagingLong</code>	<code>mapping</code>	<code>summingInt</code>
<code>composite</code>	<code>maxBy</code>	<code>summingLong</code>
<code>counting</code>	<code>minBy</code>	<code>varianceOf</code>
<code>filtering</code>		

- 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.

3 Textual Query Language Extension

- Reference 32
- Usage and Best Practice 39

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 <code>DatasetReader</code> is required to read the contents from a <code>Dataset</code> .
2	A <code>TqlEnvironment</code> is created, which takes as parameter the <code>DatasetReader</code> ,
3	A <code>String</code> alias for referencing the <code>Dataset</code> in a query,
4	and var-args of <code>CellDefinition</code> , which defines the input schema used for processing the <code>Record</code> instances of the <code>Dataset</code> . Another variant uses a <code>Set</code> of cell definitions.
5	The <code>query</code> method is used to submit a TQL query string. Within that query string the alias of the <code>TqlEnvironment</code> instance is used to refer to the <code>Dataset</code> . The query takes all cell definitions into account that are defined in the input schema. If the query is valid, the <code>query</code> method returns a <code>Result</code> instance. The <code>Result</code> instance provides a <code>stream</code> method which delivers a <code>ResultStream</code> instance.
6	As <code>ResultStream</code> extends the <code>java.util.stream.Stream</code> 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 <code>CellCollection</code> , which is a collection of <code>Cell</code> 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 <code>resolveAmbiguousNames</code> 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 <code>as</code> introduces an alias for a <code>CellDefinition</code> . 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 <code>TqlEnvironment</code> instance, the key can be included by calling <code>includeKeys</code> 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 `TqlEnvironment` 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 `splititerator`, 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".

Inclusion of Record Key

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 <code>CellDefinition</code> instances of the result schema.
2	Using stream operations, the query results are consumed and converted into a <code>rows</code> 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 <code>NULL</code> , 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 <code>Optional.empty()</code> 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 39.

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 24.

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 24 .

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.