SEP-30: Support Updates in Table API

Status

Current state: Accepted

Discussion thread: https://www.mail-archive.com/dev@samza.apache.org/msg09162.html

JIRA: blocked URLSAMZA-2709 - Adding partial updates to Samza Table API

Released:

Motivation

Table in Samza is an abstraction for a data source that supports random access by key. A table could be a remote data-store, Couchbase, for example, or a local InMemory or RocksDb backed store. The Samza table API [1] currently supports gets, puts and deletes. Partial updates to existing records is a commonly requested feature in the current Table API and is supported by many stores. This document describes the proposed approach to provide support for partial updates in Table API.

Current State

Let's first start with discussing the key interfaces of the Table API design-

- ***Table***: At its core, Table interface represents a dataset that is accessible by a key. Table access can be asynchronous or synchronous. There are three broad categories of tables: local, remote and hybrid.
- ***ReadWriteTable***: Interface that represents a read-write table. It implements Table.
- ***RemoteTable:*** Provides a unified abstraction for Samza applications to access any remote data store through stream-table join.
- ***TableReadFunction & TableWriteFunction****:* Remote Table implementations access new types of stores by writing pluggable I/O "Read/Write" functions(TableReadFunction and TableWriteFunction interfaces). TableWriteFunction typically supports put and delete operations only. *Update is not currently supported.*

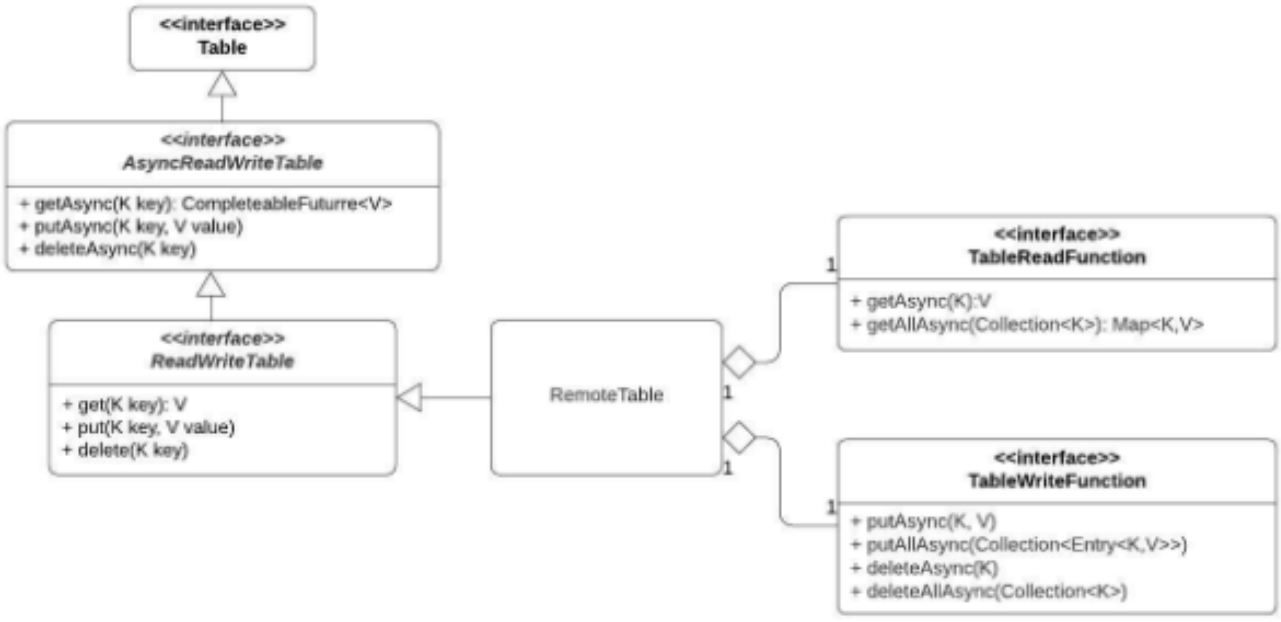

Table API Class Diagram

The sample code snippet shows a sample write to a Remote Table in Samza high level API.

```
Table<KV<Integer, Profile>> table = appDesc.getTable(desc);
appDesc.getInputStream("PageView", new NoOpSerde<PageView>())
       .map(new MyMapFunc())
       .join(table, new MyJoinFunc())
       .sendTo(anotherTable);
```

MessageStream's .sendTo method allows sending messages in a *MessageStream* to a *Table.* Under the hood, *sendTo* creates *SendToTableOperatorSpec* in the OperatorGraph which in turn is translated *SendToTableOperatorImpl*. *SendToTableOperatorImpl* is the implementation of the send-to-table operator which writes to a table by calling *ReadWriteTable's putAsync. ReadWriteTable's putAsync* call is in turn delegated to *TableWriteFunction's putAsync* method.

TableWriteFunction implementing classes typically have distinct genetic type parameters K, V specific to the table. V is the type of the record stored in the remote data store. Partial update record type is not always of the same type as the write record. Due to this type constraint, it will not be possible to change *putAsync* in Table API to support updates as well.

Proposed Solution

In order to support Partial updates, we will need to add an update API in *ReadWriteTable* and related interfaces. Update is a variant of write but *sometimes* works with a different record type when compared to Write record type. *AsyncReadWriteTable* works with generic KV where K is the key type and V is the value type of data in the Table. We need to add a generic type U to represent an update type. Adding another generic type parameter to denote an update is a backward incompatible change and would result in changes all across the Table API. The type K, V, U for a given table will be fixed.

Samza Table API Changes with Partial Update

The following changes have to be made:

- Add new *update* methods to Table API interfaces- *AsyncReadWriteTable & TableWriteFunction*
- Add *sendTo* method with *UpdateContract* to *MessageStream* API. This will be used to send updates to a table
- *UpdateContract* defines the update contract that is used in the send-to-with-update operation
- Create a new operator spec and implementation for a "send update to table" operation on a MessageStream
  - `SendToTableWithUpdateOperatorSpec`
  - SendToTableWithUpdateOperatorImpl: Will attempt to send updates using Table's *updateAsync* method. Similar to *SendToTableOperatorImpl* where writes are done using *putAsync* method.
- *UpdateMessage* class to represent an update and a default value pair instead of using KV (discussed in detail below)

AsyncReadWriteTable

```
public interface AsyncReadWriteTable<K, V, U> extends Table {
   ..
   ..
   /**
   * Asynchronously updates an existing record for a given key with the specified update.
   *
   * @param key the key with which the specified {@code value} is to be associated.
   * @param update the update applied to the record associated with a given {@code key}.
   * @param args additional arguments
   * @throws NullPointerException if the specified {@code key} is {@code null}.
   * @return CompletableFuture for the operation
   */
  CompletableFuture<Void> updateAsync(K key, U record, Object … args);

  /**
   * Asynchronously updates the existing records for the given keys with their corresponding updates.
   *
   * @param updates the key and update mappings.
   * @param args additional arguments
   * @throws NullPointerException if any of the specified {@code entries} has {@code null} as key.
   * @return CompletableFuture for the operation
   */
  CompletableFuture<Void> updateAllAsync(List<Entry<K, U>> records, Object … args);
}
```

TableWriteFunction

```
public interface TableWriteFunction<K, V, U> extends TableFunction {
  /**
   * Asynchronously update the record with specified {@code key} and additional arguments.
   * This method must be thread-safe.
   *
   * If the update operation failed due to the an existing record missing for the key, the implementation can
return
   * a future completed exceptionally with a {@link RecordNotFoundException} which will
   * allow to Put a default value if one is provided.
   *
   * @param key key for the table record
   * @param update update record for the given key
   * @return CompletableFuture for the update request
   */
  CompletableFuture<Void> updateAsync(K key, U update);

  /**
   * Asynchronously update the record with specified {@code key} and additional arguments.
   * This method must be thread-safe.
   *
   * If the update operation failed due to the an existing record missing for the key, the implementation can
return
   * a future completed exceptionally with a {@link RecordNotFoundException} which will
   * allow to Put a default value if one is provided.
   *
   * @param key key for the table record
   * @param update update record for the given key
   * @param args additional arguments
   * @return CompletableFuture for the update request
   */
  default CompletableFuture<Void> updateAsync(K key, U update, Object ... args) {
    throw new SamzaException("Not supported");
  }

  /**
   * Asynchronously updates the table with {@code records} with specified {@code keys}. This method must be
thread-safe.
   * The default implementation calls updateAsync for each entry and return a combined future.
   * @param records updates for the table
   * @return CompletableFuture for the update request
   */
  default CompletableFuture<Void> updateAllAsync(Collection<Entry<K, U>> records) {
    final List<CompletableFuture<Void>> updateFutures = records.stream()
        .map(entry -> updateAsync(entry.getKey(), entry.getValue()))
        .collect(Collectors.toList());
    return CompletableFuture.allOf(Iterables.toArray(updateFutures, CompletableFuture.class));
  }
}
```

MessageStream

```
public interface MessageStream<M> {
/**
  * Allows sending update messages in this {@link MessageStream} to a {@link Table} and then propagates this
  * {@link MessageStream} to the next chained operator. The type of input message is expected to be {@link KV},
  * otherwise a {@link ClassCastException} will be thrown. The value is an UpdateMessage- update and default
value.
  * Defaults are optional and can be used if the Remote Table integration supports inserting a default through
PUT in
  * the event an update fails due to an existing record being absent.
  * <p>
  * Note: The update will be written but may not be flushed to the underlying table before its propagated to the
  * chained operators. Whether the message can be read back from the Table in the chained operator depends on
whether
  * it was flushed and whether the Table offers read after write consistency. Messages retain the original
partitioning
  * scheme when propagated to next operator.
  *
  * @param table the table to write messages to
  * @param contract Update contract which defines how the update will be performed
  * @param <K> the type of key in the table
  * @param <V> the type of record value in the table
  * @param <U> the type of update value for the table
  * @return this {@link MessageStream}
  */
 <K, V, U> MessageStream<KV<K, UpdateMessage<U, V>>> sendTo(Table<KV<K, V>> table, UpdateContract contract);
}
```

Handling First Time Updates

While partial updates are intended to update existing records, there will be certain cases which require support for first-time partial updates i.e update to a record which doesn't exist. To account for such cases, the design needs to have a provision to optionally provide a default record which can be PUT in the absence of an existing record. The update can then be applied on top of the default record.

The approach introduces an *UpdateMessage* class which captures the update and an optional default. *sendTo* operator which sends updates to a table is designed to take a key (which uniquely identifies a record) and the *UpdateMessage* as the value. The user is also required to pass a UpdateContract. UPDATE_WITH_DEFAULTs param with *MessageStream's sendTo* to enable this.

SendToTableWithUpdateOperatorImpl is the implementation of the send-update-to-table operator and whether this operator supports first-time partial update entirely depends on the Remote store's implementation of the TableWriteFunction. This approach introduces a *RecordNotFoundException* which is a custom exception to be thrown in the *updateAsync* method of *TableWriteFunction* if the update fails due to an existing record not present for the key. If the *SendUpdateToTableOperatorImpl* operator encounters this exception, it attempts to PUT a default (if one is provided) and then applies an update on top of it.

UpdateMessage

```
/**
 * Represents an update and an optional default record to be inserted for a key,
 * if the update is applied to a non-existent record.
 *
 * @param <U> type of the update record
 * @param <V> type of the default record
 */
public final class UpdateMessage<U, V> {
  private final U update;
  @Nullable private final V defaultValue;

  public static <U, V> UpdateMessage<U, V> of(U update, @Nullable V defaultValue) {
    return new UpdateMessage<>(update, defaultValue);
  }

  public static <U, V> UpdateMessage<U, V> of(U update) {
    return new UpdateMessage<>(update, null);
  }

  private UpdateMessage(U update, V defaultValue) {
    this.update = update;
    this.defaultValue = defaultValue;
  }

  public U getUpdate() {
    return update;
  }

  public V getDefault() {
    return defaultValue;
  }
}
```

SendToTableWithUpdateOperatorImpl

```
public class SendToTableWithUpdateOperatorImpl<K, V, U>
    extends OperatorImpl<KV<K, UpdateMessage<U, V>>, KV<K, UpdateMessage<U, V>>> {
  private static final Logger LOG = LoggerFactory.getLogger(SendToTableWithUpdateOperatorImpl.class);

  private final SendToTableWithUpdateOperatorSpec<K, V, U> spec;
  private final ReadWriteTable<K, V, U> table;

  public SendToTableWithUpdateOperatorImpl(SendToTableWithUpdateOperatorSpec<K, V, U> spec, Context context) {
    this.spec = spec;
    this.table = context.getTaskContext().getTable(spec.getTableId());
    if (context.getTaskContext().getTable(spec.getTableId()) instanceof RemoteTable) {
      RemoteTable<K, V, U> remoteTable = (RemoteTable<K, V, U>) table;
      if (remoteTable.getBatchProvider() instanceof CompactBatchProvider) {
        throw new SamzaException("Batching is not supported with Compact Batches for partial updates");
      }
    }
  }

  @Override
  protected void handleInit(Context context) {
  }

  @Override
  protected CompletionStage<Collection<KV<K, UpdateMessage<U, V>>>> handleMessageAsync(KV<K, UpdateMessage<U,
V>> message,
      MessageCollector collector, TaskCoordinator coordinator) {
    final UpdateContract contract = spec.getContract();
    final CompletableFuture<Void> updateFuture = table.updateAsync(message.getKey(), message.getValue().
```

```
getUpdate());

    return updateFuture
        .handle((result, ex) -> {
          if (ex == null) {
            // success, no need to Put a default value
            return false;
          } else if (ex.getCause() instanceof RecordNotFoundException && message.getValue().hasDefault()) {
            // If an update fails for a given key due to a RecordDoesNotExistException exception thrown and
            // a default is provided and the UpdateContract is set to UPDATE_WITH_DEFAULTS, then attempt
            // to PUT a default record for the key and then apply the update.
            if (contract == UpdateContract.UPDATE_WITH_DEFAULTS) {
              return true;
            } else {
              throw new SamzaException("Put default failed for update as the Update contract was set to " +
contract +
                  ". Please use UpdateContract.UPDATE_WITH_DEFAULTS instead.");
            }
          } else {
            throw new SamzaException("Update failed with exception: ", ex);
          }
        })
        .thenCompose(shouldPutDefault -> {
          if (shouldPutDefault) {
            final CompletableFuture<Void> putFuture = table.putAsync(message.getKey(), message.getValue().
getDefault());
            return putFuture
                .exceptionally(ex -> {
                  LOG.warn("PUT default failed due to an exception. Ignoring the exception and proceeding with
update. "
                          + "The exception encountered is: ", ex);
                  return null;
                })
                .thenCompose(res -> table.updateAsync(message.getKey(), message.getValue().getUpdate()));
          } else {
            return CompletableFuture.completedFuture(null);
          }
        }).thenApply(result -> Collections.singleton(message));
  }

  @Override
  protected void handleClose() {
    table.close();
  }

  @Override
  protected OperatorSpec<KV<K, UpdateMessage<U, V>>, KV<K, UpdateMessage<U, V>>> getOperatorSpec() {
    return spec;
  }
}
```

Update support in different Table types

Batching Table

Batching tables supports batching of table operations. The key classes/interfaces are:

- Batch: Maintains a sequences of operations
- BatchProvider: Creates a batch
- Operation: Represents a table operation that can be batched
- BatchProcessor: Places operations into batches
- BatchHandler: Defines how batches will be handled

Current components are designed to work with K, V generic types of a Table and would need to be modified to include update generic type U as well. The signature of the Operation interface will be changed to add *getUpdate* as well. In addition, an *UpdateOperation* class representing an Update table operation will be created. BatchHandler and BatchProcessor will be modified to handle UpdateOperations.

Operation

```
/**
 * Interface for table operations that can be batched.
 *
 * @param <K> The key type associated with the operation.
 * @param <V> The value type associated with the operation.
 * @param <U> The update type associated with the operation.
 */
public interface Operation<K, V, U> {
  /**
   * @return The key associated with the operation.
   */
  K getKey();

  /**
   * @return The value associated with the operation.
   */
  V getValue();

  /**
   * @return The update associated with the operation.
   */
  U getUpdate();

  /**
   * @return The extra arguments associated with the operation.
   */
  Object[] getArgs();
}
```

UpdateOperation

```
/**
 * Update operation.
 *
 * @param <K> The type of the key.
 * @param <U> The type of the update
 */
public class UpdateOperation<K, V, U> implements Operation<K, V, U> {
  final private K key;
  final private U update;
  final private Object[] args;

  public UpdateOperation(K key, U update, Object ... args) {
    Preconditions.checkNotNull(key);
    Preconditions.checkNotNull(update);
    Preconditions.checkNotNull(args);
    this.key = key;
    this.update = update;
    this.args = args;
  }

  /**
   * @return The key to be updated in the table.
   */
  @Override
  public K getKey() {
    return key;
  }

  /**
   * @return null.
   */
  @Override
  public V getValue() {
    return null;
  }

  /**
   * @return The Update to be applied to the table for the key.
   */
  @Override
  public U getUpdate() {
    return update;
  }

  @Override
  public Object[] getArgs() {
    return args;
  }
}
```

Retriable Table

AsyncRetriableTable currently uses *Failsafe* library for handling retries. Reads (get) and Writes (puts, deletes) each currently have a *RetryPolicy* and metrics associated. The metrics reported are retry count, success count, perm failure count and retry timer. We will reuse the Write RetryPolicy for updates as well and metrics reported would be the same as for writes.

Rate Limited Table

Rate limited table currently uses Guava's RateLimiter, one each for read and write operations uniquely identified by tags. We will reuse the write rate limiter for write for updates as well.

Local Table

Backed by Samza's KeyValueStore. It will not support updates as the underlying store doesn't support updates.

Caching Table

Caching tables will not support updates.

Partial Update Code Example

A simple code example below for writing updates to a table using Samza table API:

Code Example

```
final RemoteTableDescriptor outputTableDesc = new RemoteTableDescriptor<Integer, EnrichedPageView,
EnrichedPageView>("enriched-page-view-table-1");

final Table<KV<Integer, Profile>> joinTable = appDesc.
   getTable(outputTableDesc);

appDesc.getInputStream(isd)
 .map(pv -> new KV<>(pv.getMemberId(), pv))
 .join(joinTable, new PageViewToProfileJoinFunction())
 .map(m -> new KV(m.getMemberId(), UpdateMessage.of(m, m)))
 .sendTo(outputTable, UpdateContract.UPDATE_WITH_DEFAULTS);
```

Test Plan

- Test plan would include unit tests to capture changes to the Table API and to the operator graph
- Add tests for update in different table types: *TestBatchTable*, *TestAsyncRetriableTable*, *TestRemoteTable*, *TestAsyncRateLimitedTable*
- Update End to end tests to test *sendUpdateTo* operator: *TestRemoteTableEndToEnd*, *TestRemoteTableWithBatchEndToEnd*
- Samza remote store integrations will be tested with unit tests and test flows

Rollout

The plan is to release this feature with Samza 1.7 release. The Table API changes are backward incompatible as *AsyncReadWriteTable* will now add a new generic type U to indicate an update in the class definition. Table integrations will have to be updated as well.

References

1. Samza Table API: https://samza.apache.org/learn/documentation/1.0.0/api/table-api.html