FLIP-527: State Schema Evolution for RowData

Motivation

Flink applications often need to evolve their input schemas as business requirements change. However, users of the Table API or SQL frequently encounter serialization compatibility errors during state restoration even when the SQL logic remains unchanged. This occurs because Flink's current state migration mechanism does not support schema evolution for RowData types, particularly when dealing with nested structures. As a result, users are unable to make safe, backward-compatible changes such as:

- Adding nullable fields to existing structures
- Reordering fields within a row while preserving field names
- Evolving nested Row structures

This limitation affects many production workloads where RowData is the core representation of state. Developers are forced to either preserve outdated input schemas or discard state and reprocess from scratch whenever schema changes are required.

Problem Example

This FLIP proposes state schema evolution support only when:

- The SQL logic remains unchanged and the input schema evolves in a backward-compatible way, or
- The SQL changes, but the underlying state mapping remains valid

Consider a Flink SQL job that stores metadata in a nested row structure:

Initial Schema

```
CREATE TABLE Events (
  eventId BIGINT,
  metadata ROW<
    userId INT,
    timestamp BIGINT,
    deviceType STRING
  >
) WITH (...);
```

Evolved Schema Example (Supported)

```
CREATE TABLE Events (
  eventId BIGINT,
  metadata ROW<
    deviceType STRING,       -- Reordered field
    location STRING,         -- New nullable field
    userId INT,              -- Original field
    timestamp BIGINT,        -- Original field
    appVersion STRING,       -- New nullable field
    sessionId BIGINT         -- New nullable field
  >
) WITH (...);
```

Current Behavior

When the job is redeployed with this updated schema, it fails with an error:

```
Caused by: org.apache.flink.util.StateMigrationException: The new state serializer
(...) must not be incompatible with the old state serializer(...).
    at
org.apache.flink.contrib.streaming.state.RocksDBKeyedStateBackend.updateRestoredSta
teMetaInfo(...)
    ...
```

This proposal introduces backward-compatible schema evolution for `RowData`, allowing jobs to recover from savepoints even when compatible schema changes are made.

Unsupported Schema Changes

The following changes would be rejected with clear error messages:

Removing Fields

```
CREATE TABLE Events (
  eventId BIGINT,
  metadata ROW<
    deviceType STRING,
    timestamp BIGINT         -- userId field removed
  >
) WITH (...);
```

Changing Field Types

```
CREATE TABLE Events (
  eventId BIGINT,
  metadata ROW<
    userId INT,
    timestamp TIMESTAMP,     -- Changed from BIGINT to TIMESTAMP
    deviceType STRING
  >
) WITH (...);
```

SQL Logic Changes Affecting State Mapping

```
-- Original:
SELECT a, b, SUM(c) FROM table GROUP BY a, b  -- buffer: agg0_max

-- Modified:
SELECT a, b, SUM(d), SUM(c) FROM table GROUP BY a, b  -- buffers: agg0_max,
agg1_max
```

In such cases, state alignment is not guaranteed as the buffer for SUM(c) changes from agg0_max to agg1_max. Users must disable schema migration unless they have verified that state mapping remains correct.

Summary

This FLIP introduces opt-in, backward-compatible schema migration support for `RowData` state.

When Schema Migration Works

- Adding new nullable fields (initialized to null)
- Reordering existing fields (field names will be used for mapping)
- Evolving nested `Row` types following the above rules

When Schema Migration Doesn’t Work

- Removing fields
- Renaming fields (name-based mapping fails)
- Changing field types (e.g., INT → STRING)

Unsupported changes trigger restore-time validation errors.

Public Interfaces

The proposal introduces the following changes to public interfaces:

- New Configuration Option
    - A new configuration option `state.schema-evolution.enable` will be introduced to give users control over when state schema evolution is applied.
    - This option is disabled by default for safety.
- Enhancements to `TypeSerializerSnapshot` Interface
    - New method: `migrateState()` - Handles full state migration
    - New method: `migrateElement()` - Handles individual element migration
- `RowDataSerializer` Enhancements
    - Add `String[] fieldNames` for name-based field mapping (lightweight alternative to storing full original RowType)
    - Enhanced `RowDataSerializerSnapshot` to support:
        - Adding nullable fields
        - Reordering fields using field names
        - Evolving nested structures using the same rules

Proposed Changes

The implementation introduces specialized state migration support for RowData types by:

- Enhanced `TypeSerializerSnapshot` Interface:

```
public interface TypeSerializerSnapshot<T> {
    // Existing methods...

    // New methods:
    default void migrateState(TypeSerializer<T> priorSerializer,
                             TypeSerializer<T> newSerializer,
                             DataInputDeserializer serializedOldValueInput,
                             DataOutputSerializer
serializedMigratedValueOutput) throws IOException {
        T value = priorSerializer.deserialize(serializedOldValueInput);
        newSerializer.serialize(value, serializedMigratedValueOutput);
    }

    default void migrateElement(TypeSerializer<T> priorSerializer,
                              TypeSerializer<T> newSerializer,
                              T element,
                              DataOutputSerializer
serializedMigratedValueOutput) throws IOException {
        newSerializer.serialize(element, serializedMigratedValueOutput);
    }
}
```

- Updated RocksDB State Backend:
    - Modify `AbstractRocksDBState` to use the new migration methods
    - Update `RocksDBListState` for element-wise RowData migration
    - Update `RocksDBMapState` for value-specific RowData migration
- Backward-Compatible Schema Evolution Support:
    - Add `case ROW` in `InternalSerializers`

```
case ROW:
    return new RowDataSerializer((RowType) type);
```

    - Add a new field `fieldNames` of type `String[]` to `RowDataSerializer` to store field names for name-based mapping during schema evolution
    - Enhanced `RowDataSerializerSnapshot` in `RowDataSerializer`
        - Support for adding nullable fields (defaulting to null for existing records)
        - Support for field reordering based on field names
        - Support for nested structure evolution following the same compatibility rules

```
public class RowDataSerializerSnapshot implements TypeSerializerSnapshot<RowData>
{
        // Existing methods...

        // Enhanced compatibility resolution to support schema evolution
        @Override
        public TypeSerializerSchemaCompatibility<RowData>
resolveSchemaCompatibility(
                TypeSerializer<RowData> newSerializer) {
            // Implementation checks for compatible schema changes:
            //   - Adding nullable fields
            //   - Field reordering with preserved names
            //   - Nested structure changes following the same rules
            // Returns compatibleAfterMigration() for supported changes
        }

        // Specialized implementation for RowData migration
        @Override
        public void migrateState(
                TypeSerializer<RowData> priorSerializer,
                TypeSerializer<RowData> newSerializer,
                DataInputDeserializer in,
                DataOutputSerializer out) throws IOException {
            RowDataSerializer oldRowSerializer = (RowDataSerializer)
priorSerializer;
            RowDataSerializer newRowSerializer = (RowDataSerializer)
newSerializer;

            // 1. Deserialize the old row data
            RowData oldRowData = oldRowSerializer.deserialize(in);

            // 2. Create a new GenericRowData with the new schema's arity
            GenericRowData newRowData = new
GenericRowData(newRowSerializer.getArity());

            // 3. Map fields from old to new positions (handling reordering)
            int[] fieldMapping = createFieldMapping(
                oldRowSerializer.getFieldTypes(),
                oldRowSerializer.getFieldNames(),
                newRowSerializer.getFieldTypes(),
                newRowSerializer.getFieldNames());

            // 4. Copy existing fields to their new positions, set nulls for added
fields
            for (int newPos = 0; newPos < newRowSerializer.getArity(); newPos++) {
                int oldPos = fieldMapping[newPos];
                if (oldPos >= 0 && oldPos < oldRowSerializer.getArity()) {
                    // Field exists in old schema, copy the value
                    newRowData.setField(newPos, oldRowData.isNullAt(oldPos) ?
                        null : oldRowData.getField(oldPos));
                } else {
                    // New field, set to null
                    newRowData.setField(newPos, null);
                }
            }

            // 5. Serialize the migrated row data
            newRowSerializer.serialize(newRowData, out);
        }

        // Similar implementation for element-wise migration
        @Override
        public void migrateElement(
                TypeSerializer<RowData> priorSerializer,
                TypeSerializer<RowData> newSerializer,
                RowData element,
                DataOutputSerializer out) throws IOException {
            // Similar logic to migrateState but starts with already deserialized
element
            // Creates new row with proper schema, maps fields, handles nulls for
new fields
        }

        // Helper method to create field mapping between schemas
        private int[] createFieldMapping(
                LogicalType[] oldTypes, String[] oldNames,
                LogicalType[] newTypes, String[] newNames) {
            // Returns mapping array where result[newIndex] = oldIndex
            // Uses name-based matching when names are available
            // Handles position-based fallback when names aren't available
        }
    }
```

- Documentation. Add new user-facing documentation covering:
    - When schema evolution is supported vs. not supported
    - The required configuration (`state.schema-evolution.enable=true`)
    - Step-by-step instructions for restoring with evolved schemas
    - Examples for supported and unsupported schema changes

User Workflow

1. Enable migration: Set `state.schema-evolution.enable=true` in job config
2. Take a savepoint of the current job
3. Update the application with the new input schema (with only supported changes)
4. Restore from the savepoint → Flink will perform automatic migration if the change is safe

Compatibility, Deprecation, and Migration Plan

Impact on existing users:

- The changes are backward compatible and don't modify existing behavior
- No migration required for users who don't need schema evolution
- Users who previously couldn't evolve their RowData schemas can now do so with a savepoint

Compatibility guarantees:

- Only backward-compatible schema changes are supported:
    - Adding nullable fields (non-nullable fields still unsupported)
    - Reordering fields with preserved names
    - Changes to nested structures following these same rules
- Others will fail safely with clear error messages

Test Plan

The implementation is tested through:

- Unit tests covering various schema evolution scenarios:
  - Adding nullable fields at different positions
  - Reordering fields
  - Nested structure evolution
  - Combination of multiple compatible changes
- Integration tests verifying (Add `SchemaEvolutionWithStateITCase`):
  - End-to-end behavior with savepoints
  - Proper rejection of incompatible changes

The tests ensure that:

- Compatible schema changes succeed
- Incompatible schema changes are rejected with clear errors
- Migration works correctly with the RocksDB state backend

Rejected Alternatives

- Create a custom state serialization instead of updating the existing `RowDataSerializer`:
  - Considered creating a separate `SchemaEvolutionRowDataSerializer` that would handle schema evolution
  - Rejected for several reasons:
    - Would require users to explicitly use this serializer instead of getting automatic support
    - Would introduce divergent code paths and potential maintenance issues
    - Would not integrate well with Flink's existing serialization framework
  - The chosen approach of enhancing the existing serializers provides a more seamless experience and maintains backward compatibility