FLIP-188: Introduce Built-in Dynamic Table Storage

Status

<table>
<thead>
<tr>
<th>Discussion thread</th>
<th><a href="https://lists.apache.org/thread/tqyn1cro5ohi3c3kjb1zvxbo03sofn7">https://lists.apache.org/thread/tqyn1cro5ohi3c3kjb1zvxbo03sofn7</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vote thread</td>
<td></td>
</tr>
<tr>
<td>JIRA</td>
<td><img src="FLINK-25152.png" alt="FLINK-25152 - FLIP-188: Introduce Built-in Dynamic Table Storage" /></td>
</tr>
<tr>
<td>Release</td>
<td>1.15</td>
</tr>
</tbody>
</table>

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

- Status
- Motivation
- Proposal
- Public Interfaces
  - Example
    - SQL Statements
      - CREATE
      - DROP
      - COMPACT
      - READING
      - INSERT
      - DESCRIBE
    - Configuration
      - Session Options
      - Table Options
    - Bucket
    - Checkpoint
    - Concurrent Write
    - Retention
      - Log Retention
      - Data Retention
    - Interfaces for Table
- Proposed Design
  - LogStore
    - Format
    - Consistency & Visibility
    - Changelog mode
    - Optimize Upsert mode
  - FileStore
    - Overview
    - Write Process
    - Compaction
    - Query Process
    - Support Changelog
    - Query Pushdown
- Rejected Alternatives
  - Using Hudi
  - Add Primary Key
  - Change Buckets
- Implementation Plan

Motivation

Streaming analytics is one the most important use cases among all SQL usage. Users are relying on the streaming nature of Flink to deliver subsecond end to end latency. The typical architecture for such use case is like following:
The message queue (mostly going with Kafka) will be used in both source & intermediate stages in this pipeline, to guarantee the latency stay within seconds. There will also be a real-time OLAP system receiving processed data in streaming fashion and serving user’s ad-hoc queries.

Everything works fine as long as users only care about the aggregated results. But when users start to care about the intermediate data, they will immediately hit a blocker: Intermediate kafka tables are not queryable.

Therefore, users could dual write data to Hudi or Clickhouse so that the intermediate tables can be queried. Besides from this, most message queue is known expensive to store too much data, therefore users either relying on a TTL to delete outdated data or export the data out from the message queue to some more cost friendly storage, like iceberg.

For now, let's focus on the dual write solution and try to write the SQLs:

```sql
CREATE TEMPORARY VIEW intermediate_table AS
SELECT
    A.order_id,
    A.auction_id,
    B.category_id,
    A.trans_amount,
    A.create_time
FROM orders A LEFT JOIN category_dim B
ON A.auction_id = B.auction_id;

-- Or create a Upsert-Kafka Table to accept changelog
CREATE TABLE kafka_intermediate_table_0 (order_id BIGINT,
                                          auction_id BIGINT,
                                          category_id BIGINT,
                                          trans_amount BIGINT,
                                          create_time BIGINT
);
```

---

**Diagram:**

The diagram shows the pipeline process with sources, intermediate results, joins, and aggregations leading to aggregated results. The message queue (mostly with Kafka) guarantees latency within seconds. The real-time OLAP system processes data in streaming fashion.

---

**Notes:**

- Users care about aggregated results until they start caring about the intermediate data.
- Dual write to Hudi or Clickhouse allows querying intermediate data.
- Message queues are expensive for storing too much data; use TTL or exporting to more cost-effective storage like iceberg.

---

**SQL Example:**

- CREATE TEMPORARY VIEW for intermediate data.
- CREATE TABLE for the Kafka intermediate table with indexes and columns.

---

**Image:**

The image illustrates the pipeline flow, from sources to intermediate results, through joins and aggregations, leading to aggregated results.
category_id BIGINT,
trans_amount BIGINT,
create_time TIMESTAMP,
PRIMARY KEY (order_id) NOT ENFORCED
) WITH {
  'connector' = 'upsert-kafka',
  'topic' = '...',
  'properties.bootstrap.servers' = 'localhost:9092',
  'properties.group.id' = 'testGroup',
  'key.format' = 'avro',
  'value.format' = 'avro'
};

-- Create a Kafka Table with debezium-avro to accept changelog

CREATE TABLE kafka_intermediate_table_1 (
  order_id BIGINT,
  auction_id BIGINT,
  category_id BIGINT,
  trans_amount BIGINT,
  create_time TIMESTAMP
) WITH {
  'connector' = 'kafka',
  'topic' = '...',
  'properties.bootstrap.servers' = 'localhost:9092',
  'properties.group.id' = 'testGroup',
  'format' = 'debezium-avro'
};

-- Create a Hudi Table to accept changelog

CREATE TABLE hudi_intermediate_table (
  order_id BIGINT,
  auction_id BIGINT,
  category_id BIGINT,
  trans_amount BIGINT,
  create_time TIMESTAMP,
  dt STRING,
  PRIMARY KEY (order_id) NOT ENFORCED
) PARTITIONED BY (dt) WITH {
  'connector' = 'hudi',
  'path' = '...',
  'write.precombine.field' = 'create_time',
  'table.type' = 'MERGE_ON_READ'
};

-- Insert into

INSERT INTO kafka_intermediate_table SELECT * FROM intermediate_table;

INSERT INTO hudi_intermediate_table SELECT *,
DATE_FORMAT(create_time, 'yyyy-MM-dd')
FROM intermediate_table;

-- Query: Streaming Pipeline

INSERT INTO ... SELECT ... FROM kafka_intermediate_table;

-- Query: Ad-hoc query

SELECT * FROM hudi_intermediate_table WHERE ...;

From above example and other feedback we received in various channel (community users, mailing lists, IM groups), we boiled down all the issues into following categories:

- High understanding bar for new users

1. Although Flink SQL sticks to ANSI SQL as much as possible, it’s still relatively hard for new users to understand some streaming nature. For example, what’s the differences between windowed aggregation and non-windowed aggregation, what’s the differences between the query results.
2. It’s also not easy for new users to understand all the SQL connectors, learn the capabilities and restrictions for each of those. We’ve seen many users choose csv/json/avro formats with Kafka and run into errors immediately, saying it can’t handle update /delete row kinds (if the query contains retractions results like aggregates), but doesn’t understand it or how to fix it.
3. It’s not clear what kind of consistency and messaging guarantees in this case, and how to tune it.
   ◦ If you already passed the beginner phase, it’s sometimes still difficult to use even for experienced users
   1. Connectors with update/delete capabilities are still relying on primary keys. It’s not easy or even possible for users to define a precise primary key for each table. For example, a left outer join could produce retraction messages without a primary key.
   2. You may also want to play around with streaming & batch unification, but don’t really know how, given the connectors are most of the time different in batch and streaming use cases. For example, what’s the best practice to do a backfilling with SQL in those cases.

* Increasing architecture complexity

1. It’s hard to choose the most suited external systems when the requirements include streaming pipelines, offline batch jobs, ad-hoc queries and even some point lookup queries.
2. Even if you already make your choice, it will definitely increase the operation and maintenance complexity. Users at least need to coordinate between the log system and file system of each table, which is error prone.

Proposal

If you have experience with Flink SQL, you might still be familiar with SQL’s basic concept: dynamic table. In short, a dynamic table is a logical concept which has two different physical representations: changelog and table. Right now, by relying on SQL connectors, users can define table which acts like one of the physical representations, but not both.

For example, users tend to use kafka to store logs and use hudi/iceberg/clickhouse as a table. We propose to introduce built-in storage support for dynamic table, a truly unified changelog & table representation, from Flink SQL’s perspective. We believe this kind of storage will improve the usability a lot. (In the future, it can support LookupTableSource too).

We want to highlight some characteristics about this storage:

* It’s a built-in storage for Flink SQL
  ◦ Improve usability issues
  ◦ Flink DDL is no longer just a mapping, but a real creation for these tables
  ◦ Masks & abstracts the underlying technical details, no annoying options
* Supports subsecond streaming write & consumption
  ◦ It could be backed by a service-oriented message queue (Like Kafka)
* High throughput scan capability
  ◦ Filesystem with columnar formats would be an ideal choice just like iceberg/hudi does.
* More importantly, in order to solve the cognitive bar, storage needs to automatically address various Insert/Update/Delete inputs and table definitions
  ◦ Receive any type of changelog, receive any type of datatype
  ◦ Table can have primary key or no primary key

Public Interfaces

Example

If we have a built-in Flink Dynamic Table, users just focus on their business logic:
-- Just business fields, primary key is not mandatory

CREATE TABLE intermediate_table (
    order_id BIGINT,
    auction_id BIGINT,
    category_id BIGINT,
    trans_amount BIGINT,
    create_time TIMESTAMP,
    dt STRING
) PARTITIONED BY (dt);

-- Insert into

INSERT INTO intermediate_table
SELECT
    A.order_id,
    A.auction_id,
    B.category_id,
    A.trans_amount,
    A.create_time,
    DATE_FORMAT(create_time, 'yyyy-MM-dd')
FROM orders A LEFT JOIN category_dim B
ON A.auction_id = B.auction_id;

-- Query: Streaming Pipeline

INSERT INTO ... SELECT ... FROM intermediate_table;

-- Query: Batch ad-hoc query

SELECT * FROM intermediate_table WHERE ...;
When creating a table, the corresponding underlying physical storage will be created. Very simple, it masks & abstracts the underlying technical details, no annoying options.

Limitation: When a partitioned table has a primary key, the primary key must contain the partitioned fields inside.

**DROP**

```sql
DROP [TEMPORARY] TABLE [IF EXISTS] [catalog_name.] [db_name.] table_name
```

When dropping a table, the corresponding underlying physical storage will be deleted.

(If the user does not drop the table through Flink, the physical storage under the table may not be deleted all the time. The user can record the response address through describe and handle it by himself)

**COMPACT**

```sql
ALTER TABLE [catalog_name.] [db_name.] table_name [PARTITION partition_spec] COMPACT
```

Compact table for high performance query. Launch a job to rewrite files. It is a synchronous operation.

**READING**

```sql
-- unbounded streaming reading (Read changes)
SET 'execution.runtime-mode' = 'streaming';
INSERT INTO ... SELECT ... FROM [catalog_name.][db_name.]table_name;

-- bounded reading (Read a snapshot)
SET 'execution.runtime-mode' = 'batch';
INSERT INTO ... SELECT ... FROM [catalog_name.][db_name.]table_name;
```

The table supports both stream reading (read changes) and high-performance batch reading.

**INSERT**

```sql
-- unbounded insert, not support OVERWRITE
INSERT INTO [catalog_name.][db_name.]table_name
  [PARTITION part_spec] [column_list] select_statement;

-- bounded insert
INSERT [ INTO | OVERWRITE ] [catalog_name.] [db_name.] table_name
  [PARTITION part_spec] [column_list] select_statement;

part_spec:
  (part_col_name1=val1 [, part_col_name2=val2, ...])

column_list:
  (col_name1 [, column_name2, ...])
```

Users can write any type of changelog with any SQL.

The changes by Batch jobs will be tracked by default. But sometimes, like in the revision of the old partition of the whole pipeline, the state of the downstream stream job may have expired long ago. What we need is the batch pipeline.
In this case, we need to close the changes tracking of this writing, batch job will not produce changes to downstream stream jobs. (re-processing)

```sql
INSERT INTO [catalog_name.][db_name.][table_name] /*+ OPTIONS('change-tracking' = 'false') */ ...
```

**DESCRIBE**

```sql
DESCRIBE TABLE EXTENDED [catalog_name.][db_name.][table_name] [PARTITION partition_spec]
```

**DESCRIBE TABLE EXTENDED output:**

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>catalog.database.tableName</td>
</tr>
<tr>
<td>log.system</td>
<td>String</td>
<td>the log system</td>
</tr>
<tr>
<td>log.kafka.bootstrap.servers</td>
<td>Map</td>
<td>Kafka brokers</td>
</tr>
<tr>
<td>log.retention</td>
<td>Duration</td>
<td>how long changes log will be kept</td>
</tr>
<tr>
<td>file.path</td>
<td>String</td>
<td>File path</td>
</tr>
<tr>
<td>log.kafka.topic</td>
<td>String</td>
<td>topic of Kafka</td>
</tr>
<tr>
<td>file.format</td>
<td>String</td>
<td>format for file</td>
</tr>
<tr>
<td>bucket</td>
<td>Integer</td>
<td>bucket number</td>
</tr>
<tr>
<td>change-tracking</td>
<td>Boolean</td>
<td>does this table tracking changes</td>
</tr>
</tbody>
</table>

**DESCRIBE TABLE EXTENDED ... PARTITION output:**

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition</td>
<td>String</td>
<td>partition spec</td>
</tr>
<tr>
<td>file.path</td>
<td>String</td>
<td>path of this partition</td>
</tr>
<tr>
<td>num-files</td>
<td>Integer</td>
<td>file number</td>
</tr>
</tbody>
</table>

**DESCRIBE TABLE EXTENDED** without partition definition output above columns too except partition.

**Configuration**

**Session Options**

In every table environment, the `TableEnvironment.getConfig` offers options for configuring the current session.

We put necessary configurations in the session configuration to avoid the need for users to configure each individual table.

<table>
<thead>
<tr>
<th>Key</th>
<th>Default</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table-storage.log.system</td>
<td>kafka</td>
<td>String</td>
<td>Log system. Now only Kafka in the MVP.</td>
</tr>
<tr>
<td>table-storage.log.kafka. bootstrap.servers</td>
<td>(none)</td>
<td>String</td>
<td>Kafka brokers. eg: localhost:9092</td>
</tr>
<tr>
<td>table-storage.log.retainion</td>
<td>(none)</td>
<td>Duration</td>
<td>It means how long changes log will be kept. The default value is from the log system cluster.</td>
</tr>
<tr>
<td>Key</td>
<td>Default</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>log.scan</td>
<td>full</td>
<td>String</td>
<td>Specifies the scan startup mode for log consumer.</td>
</tr>
<tr>
<td>log.scan.timestamp-mills</td>
<td>none</td>
<td>Long</td>
<td>Optional timestamp used in case of &quot;from-timestamp&quot; scan mode.</td>
</tr>
<tr>
<td>change-tracking</td>
<td>true</td>
<td>Boolean</td>
<td>If users do not need to consume changes from the table, they can disable Change Tracking. This can reduce resource consumption.</td>
</tr>
</tbody>
</table>

**Bucket**

The record is hashed into different buckets according to the primary key (if have) or the whole row (without primary key):

Bucket is for distributed reading and writing.

Bucket and Parallelism are corresponding:

- **writing**: A single bucket can only be written by a single parallelism. But one parallelism can write to multiple buckets. So the max working parallelism of the sink will not be bigger than the bucket number.
- **reading**: In general, a single bucket can only be read by a single parallelism. (Optimization: if the bucket is too large, we can consider supporting concurrent batch reading of a single bucket, which requires cutting out appropriate splits according to the max value and min value of files. The LSM supports range reading.)

More buckets:
Pros: better scalable (more distributed parallelisms)
Cons: more operation and maintenance costs

The default value of the bucket is 1, so that it can be used out of the box on the local machine.

It can be set by:

```sql
SET table-storage.bucket = 10;
CREATE TABLE ...;
```

If users want to change the bucket number, they need to delete the table and create a new table.

### Checkpoint

In the past, many users encountered the problem that the sink did not output because they did not open the checkpoint.

For the built-in dynamic table: The planner will throw an exception if the checkpoint is not turned on. (Later we can add public connector interface, including Filesystem, Hive, Iceberg, Hudi need it).

### Concurrent Write

Only a single stream writer is allowed to write data to a Dynamic table. But re-processing is allowed, so while the stream job is running, there may be another job generating a snapshot.

Write contention uses a distributed optimistic lock mechanism, for Active partition: INSERT OVERWRITE and ALTER TABLE ... COMPACT will delete files, it may conflict with the streaming job, which means that the command may fail and the user will be asked to retry. Optimism is reflected in the deletion of files. If the file to be deleted is found missing when committing, it will fail instead of locking at the beginning.

For HDFS, path renaming is used for concurrent write, if the renaming fails, it can know that the snapshotId was preempted by the another job, at which point it can recheck and generate a new snapshot.

But for object file system instead of HDFS, renaming is not work, we need catalog lock to solve commit conflicts:

```java
/**
 * An interface that allows source and sink to use global lock to some transaction-related things.
 */
@Internal
public interface CatalogLock extends AutoCloseable {

    /** Run with catalog lock. The caller should tell catalog the database and table name. */
    <T> T runWithLock(String database, String table, Callable<T> callable) throws Exception;

    /** Factory to create @link CatalogLock. */
    interface Factory extends Serializable {
        CatalogLock create();
    }
}
```

Currently, only HiveCatalog can provide this catalog lock.

And we need a interface to set lock to source&sink by catalog:
Interface

```java
/**
 * Source and sink implement this interface if they require (@link CatalogLock). This is marked as
 * internal. If we need lock to be more general, we can put lock factory into @link
 * DynamicTableFactory.Context).
 */
@Internal
public interface RequireCatalogLock {
    void setLockFactory(CatalogLock.Factory lockFactory);
}
```

Retention

Log Retention

The cost of log is generally large, so log can not save all the history of data, we provide parameters to configure the log retention time: "log.retention".

Thanks to the FileStore's data preservation, the expired data is still stored in the FileStore. By default (log.scan is full), user's stream consumption fetches all data.

So, users can set a smaller Log retention to reduce the cost in log system if the users don't need log.scan from-timestamp mode.

Data Retention

Data never expires automatically.

If there is a need for data retention, the user can choose one of the following options:

- In the SQL for querying storage, users filters the data by themselves
- Define the time partition, and users can delete the expired partition by themselves. (DROP PARTITION ...)
- In the future version, we will support "DELETE FROM" statement, users can delete the expired data according to the conditions.

Interfaces for Table

A catalog that supports built-in dynamic table needs to implement the method in the Catalog (The GenericInMemoryCatalog and HiveCatalog will implement this method):

```java
Interface

/**
 * If return true, the Table without specified connector will be translated to the Flink managed table.
 * See @link CatalogBaseTable.TableKind#MANAGED
 */
default boolean supportsManagedTable {
    return false;
}
```

We need an interface to discover the managed table factory implementation for managed table:
**Interface**

```java
/**
 * Base interface for configuring a managed dynamic table connector. The managed table factory is
 * used when there is no `{@link FactoryUtil#CONNECTOR}` option.
 */
@Internal
public interface ManagedTableFactory extends DynamicTableFactory {

    @Override
    default String factoryIdentifier() {
        return "";
    }

    /**
     * Enrich options from catalog and session information.
     * @return new options of this table.
     */
    Map<String, String> enrichOptions(Context context);

    /** Notifies the listener that a table creation occurred. */
    void onCreateTable(Context context);

    /** Notifies the listener that a table drop occurred. */
    void onDropTable(Context context);
}
```

**Table API for creating managed table:**

```java
@PublicEvolving
public class TableDescriptor {

    /** Creates a new `{@link Builder}` for a managed dynamic table. */
    public static Builder forManaged() {
        return new Builder();
    }

    ...
}
```

**Proposed Design**

Conceptually, Flink built-in tables consist of two parts, LogStore and FileStore. The LogStore would serve the need of message systems, while FileStore will play the role of file systems with columnar formats. At each point in time, LogStore and FileStore will store exactly the same data for the latest written data (LogStore has TTL), but with different physical layouts. If one remembers the concept of dynamic table in Flink SQL, this is exactly what we want to build here.

The LogStore data has faster Time-To-Live, and FileStore ensures that historical data can be queried:

- **LogStore**: Store the latest data, support second level streaming incremental consumption, rely on Kafka
  - For full support of Insert/Update/Delete and optional primary key definition
- **FileStore**: Store latest data + historical data, provide batch Ad-Hoc analysis
  - For full support of Insert/Update/Delete and optional primary key definition, we need a flexible storage structure that supports updates and custom merges, a LSM is a good choice
  - To support high performance analysis, should be columnar file format
LogStore

Log storage relies on log system. Default we use Kafka as underlying storages.

Bucket in LogStore is Kafka Partition, which means the record is hashed into different Kafka partitions according to the primary key (if have) or the whole row (without primary key).

Format

LogStore uses the open format to store record. The user can get record from the log store in a non-Flink way. By default:

- **Key**:
  - Without primary key: key is null.
  - With primary key: key is json format by default. This is controlled by 'log.key.format'.
- **Value**: Use `debezium-json` to store record with or without declaration primary key by default. This is controlled by 'log.format'.

Consistency & Visibility

By default, data is only visible after the checkpoint, which means that the logStore has transactional consistency.

If the user wants the data to be immediately visible, he/she needs to:

- Declaring the **primary key** in the table definition
- `'log.consistency' = 'eventual'`
- `'log.changelog-mode' = 'upsert'` – this is the default mode for table with primary key

When using **upsert** mode, a normalized node is generated in downstream consuming job, which will generate update_before messages for old data based on the primary key, meaning that duplicate data will be corrected to an eventual consistent state.

Changelog mode

By default, for the table with primary key, the records in the log system only contains INSERT, UPDATE_AFTER, DELETE. No UPDATE_BEFORE. A normalized node is generated in downstream consuming job, the node will store all key-value for producing UPDATE_BEFORE message.

If the user wants to see the all changes of this table or remove downstream normalized node, he/she can configure:

- `'log.changelog-mode' = 'all'`

This requires

- `'log.consistency' = 'transactional'`
- The sink query produces changes with UPDATE_BEFORE. If not, we can:
  - Throws unsupported exception in the MVP
  - In future, we can automatically add the normalize node before sink to generate required UPDATE_BEFORE messages

Optimize Upsert mode

Many users complain about upsert-kafka, where the normalized nodes in downstream consumption jobs generate a lot of state and risk state expiration.

Unlike upsert-kafka, the upsert mode preserves the complete delete message and avoids normalization for the following downstream operators:
- Upsert sink: Upsert sink only requires upsert inputs without UPDATE BEFORE.
- Join: Join for unique inputs will store records by unique key. It can work without UPDATE BEFORE.

**FileStore**

**Overview**

As a storage system supporting real-time ad-hoc analysis:

- LSM with Columnar format
- Fast update and data skipping
- High compression ratio and high performance analysis
- Partition and Bucket
- data warehouse support
- Consistency
- file management
- version control

The directory structure of FileStore on DFS is as follows:

```
  Table
    
    PART0
    PART1
    ....
    
    Bucket0  LSM DataFiles
        
        Bucket1
        ....
        
        manifest  Manifest0
        Manifest1
        
        snapshot  Snapshot0
        Snapshot1
```

Data directory description:

- Part Directory: partition directory, defined by "PARTITIONED BY" in DDL, represents a partition with the same directory name as Hive, such as "dt=2020-08-08"
- Bucket Directory: the bucket under the partition. The data falls to a bucket through hash. The bucket is an LSM composed of multiple files
- LSM datafiles: data file, abstract format, supporting orc, parquet and Avro. The record schema of data file:
  - SequenceNumber
  - ValueKind(add or delete)
  - RowData: key
  - RowData: value

Meta file description:
• Manifest file: represents how many files have been added and how many files have been deleted. It represents a change to the table. Manifest represents the incremental files of a version. The record schema of manifest file is DataFile:
  ◦ data file name
  ◦ FileKind: add or delete
  ◦ partition
  ◦ bucket
  ◦ min/max key: for file skipping
  ◦ min/max sequence number
  ◦ statistics: data file size, row count

• Snapshot file: a collection of manifest files that represents a snapshot of a table. Snapshot represents all files of a version. The record schema of snapshot file is ManifestFile:
  ◦ manifest file name
  ◦ lower/upper partition: for partition pruning
  ◦ statistics: manifest file size, addedFileCount, deleteFileCount

Write Process

1. LSM Process (Similar to LevelDB):
   a. Memtable is maintained in memory. Data is directly written to memtable. Each data has a sequence number. For the same key, data with large sequence will overwrite data with small sequence
   b. When the memtable is full or PrepareCommit, flush the memtable, sort the memtable by key + sequence number, merge the duplicate keys, and write the data to the remote file using a specific format
   c. The asynchronous thread performs LSM compactions

2. Prepare Commit
   a. Flush MemTable
   b. Commit message is: DeleteFiles and AddFiles.

3. Global Commit
   a. Get old Snapshots, if this checkpoint has been committed, just return
   b. Read the previous snapshot-$i$, write the deleteFiles and addFiles of buckets to the new manifest, and generate a new snapshot-$i+1$

Compaction

Auto compaction is in the streaming sink (writer).

We do not have independent services to compact. Independent services will bring a lot of additional design complexity, and we only need a decoupled storage in the current version. (In future, if we have a service, we can let the streaming writer be the pure append writer.)

For each LSM, there is only one streaming writer, and this writer also needs to be responsible for its compaction.

About compaction strategy, at present, we don’t have enough tests to adjust the compaction strategy. We can refer to the two mainstream strategies of rocksdb:

Leveled Compaction:

The trigger of compaction depends on three options:

• level0_file_num_compaction_trigger: When the number of level0 files exceeds this value, compact level0 files to level1
• max_bytes_for_level_base and max_bytes_for_level_multiplier: If the base is 1GB, multiplier is 5, then if the data of level1 exceeds 1GB, the compaction will be performed, and if the data of level2 exceeds 5GB, the compaction will be performed...

For Leveled Compaction, every level is a sort Run (except level0). SSTs in level1 will merge with level2, and finally form orderly SSTs in Level2, and each SST will not overlap. Leveled Compaction is the default strategy of RocksDb:

• Write amplify: bad, the data will be compacted once and once
• Read and Space amplify: good, every level no overlap

Universal Compaction:

• universal_sort_run_num_compaction_trigger: When the number of sort run exceeds this value, do compaction
• universal_max_size_amplification_percent
• universal_size_ratio

In universal mode, there are many sort runs. For R1, R2, R3, ..., Rn, each R is a sort run, R1 contains the latest data, and Rn contains the oldest data. When the preconditions are met, the following compaction is triggered in priority order:

• Compaction by Space Amplification, will do full compaction to compact all sort runs. Amplification is:
  ◦ (size(R1) + size(R2) + ... + size(Rn-1) / size(Rn)) (If the frequency of delete is similar to the frequency of insertion)
  ◦ Compaction by Individual Size Ratio: If the previous size (R1) is less than the size (R2) in a certain proportion, the default is 1%, then perform a compaction with R1 and R2. If (R1 + R2) * (100 + ratio)% 100 < R3, add R3 to the compaction.

Complicated to Leveled Compaction, Universal compaction:

• Write amplify: good, old data will not be compacted once and once
• Read and Space amplify: bad, a larger number of Sort Runs
In our scenario, writing is more. Although leveled compaction may have a better compression rate in the aggregation scenario, in the first version, we first provide universal compaction.

**Query Process**

1. **Planner**
   - a. Read the current snapshot, prune partitions according to filtering conditions, and obtain manifests to be read
   - b. Merge the deleteFiles and addFiles in manifests to generate a file list for each bucket in each partition

2. **SplitEnumerator**
   - a. Traverse the partitions to be read and generate the corresponding SourceSplit for each bucket
   - b. Filter the files to be read in the bucket according to filtering conditions, produce the files of each LSM level in SourceSplit

3. **Runtime Task**
   - a. Obtain the SourceSplit to be read, generate the MergeIterator of LSM, and read the data

**Support Changelog**

Similarly, we should shield the complexity of changelog support just like LogStore. Changelog is supported as follows:

- **DDL with Primary Key**
  - LSM Key: Primary Key
  - LSM Value: Row (All columns)

- **DDL without Primary Key**
  - LSM Key: Row (All columns)
  - LSM Value: Count, Number of occurrences of the same record
  - Count is +1 when adding and Count is -1 when deleting. Sum count when compaction and merge reading.

- **DDL with Index Key**: When there is no primary key, users can define an index key to speed up update and query. (Not in this FLIP)

**Query Pushdown**

FileStore can support more compaction strategies, help the input data to achieve the effect of lazy computation. (Not in this FLIP) For example:

- SUM Compaction: Non-key fields will be grouped by to sum aggregation.
- COALESCE Compaction: just store non-null fields, it can replace streaming join to widen the fields

**Rejected Alternatives**

**Using Hudi**

Hudi: [https://hudi.apache.org/](https://hudi.apache.org/)

Why doesn't FileStore use Hudi directly?

- Hudi aims to support the update of upsert, so it needs to forcibly define the primary key and time column. It is not easy to support all changelog types
- The update of Hudi is based on the index (currently there are BloomFilter and HBase). The data in the bucket is out of order. Every merge needs to be reread and rewritten, which is expensive. We need fast update storage, LSM is more suitable.

**Add Primary Key**

The Flink Built-in Dynamic Table supports free switching with and without primary key:

```sql
ALTER TABLE [catalog_name.][db_name.]table_name
ADD PRIMARY KEY (column_name, ...) NOT ENFORCED
```

Add primary key to a table without primary key:

- **FileStore**
  - Launch a job to rewrite files
  - If there are duplicate keys, the command will fail

- **LogStore**
  - Truncate logs: Delete current topic and create a new topic
  - Jobs that are currently consuming this table will fail
The cost of LogStore is too high, users can't continue their streaming consumption, and can only start consumption from the latest. Therefore, it is not supported at present.

Change Buckets

```sql
ALTER TABLE [catalog_name.][db_name.]table_name [PARTITION partition_spec]
   CLUSTERED INTO num_buckets BUCKETS;
```

Change Bucket number for FileStore tradeoff between scalable and performance. Launch a job to rewrite files. (The first version is not available. Users need to delete table and create a new table)

Implementation Plan

- POC branch: https://github.com/JingsongLi/flink/tree/table_storage
- Implement in dev branch, the code will not enter the master branch for the time being
  - Implement FileStore
    - Abstract Format: support ORC and Parquet
    - Implement LSM: MemStore and Compaction
    - Implement Snapshot and Manifest: Version control
  - Implement LogStore
    - Auto create Kafka Topic
    - Integrate CDC Format and Upsert Kafka
  - Integrate Flink
    - TableFactory: DynamicSource and DynamicSink
    - Integrate to Catalog
- Extended DMLs