FLIP-556: Introduce BITMAP Data Type

Motivation

Flink currently lacks a built-in data type for efficient storage and set operations on large collections of integers, such as user IDs, device IDs, or time-sliced identifiers. While exact high-cardinality deduplication is a common requirement in real-time analytics, Flink does not provide native support for compressed integer sets. As a result, users who need precise distinct counting or set algebra (e.g., intersection of user groups) are forced to integrate external libraries like RoaringBitmap[1] through UDFs or custom state serializers—an approach that is error-prone, hard to maintain, and lacks integration with Flink's type system, SQL engine, and state management.

To address this, we propose introducing a new native BITMAP type for 32-bit unsigned integers, implemented using the widely adopted RoaringBitmap, which has become the de facto standard for compressed bitmap representations and is supported by major analytical systems including ClickHouse[2], StarRocks[3], Doris[4], and PostgreSQL (via extensions)[5].

The BITMAP type provides:

- Compact storage of integer sets through adaptive compression;
- Exact deduplication with constant-time cardinality;
- Efficient logical operations (AND, OR, XOR, ...) directly on compressed data.

BITMAP Semantics

The BITMAP type represents a set of 32-bit unsigned integers, with the following semantic properties:

- 32-bit integer storage: Only integers in the logical range [0, 2^32) are supported. 64-bit integers are out of scope (see description below).
- Unsigned interpretation: Values are treated as unsigned. For example, the Java integer -1 (bit pattern 0xFFFFFFFF) is interpreted as 4294967295. This affects some logic related to ordering.
- Equality and hashing: BITMAP supports equals() and hashCode(), enabling its use in GROUP BY and DISTINCT.
- Non-comparable: BITMAP does not define a total order. It cannot be used in ORDER BY, comparison predicates, or as a key in ordered data structures.
- Nullability: BITMAP supports NULL values. NULL represents the absence of a bitmap, while an empty bitmap is a valid object with zero elements. For example:
    - BITMAP_CARDINALITY(NULL) → NULL
    - BITMAP_CARDINALITY(empty_bitmap) → 0
    - BITMAP_OR(bitmap, NULL) → NULL

        - BITMAP_OR(bitmap, empty_bitmap) → bitmap
- Casting rules:
    - Implicit casts: Not supported.
    - Explicit casts:
        - From: BINARY, VARBINARY, ARRAY<INT>;
        - To: BINARY, VARBINARY, CHAR, VARCHAR, ARRAY<INT>;
        - All other explicit casts are not supported.

The BITMAP type can be declared in DDL (e.g., CREATE TABLE t (userId BITMAP, ...)).

We limit the initial scope to 32-bit integers. This covers the vast majority of practical use cases—user/device IDs, bucketed timestamps, etc.—which can often be mapped into 32-bit space via dictionary encoding. In contrast, 64-bit Roaring implementations lack standardization and may introduce significant memory and serialization overhead in streaming workloads. Should there be strong demand, 64-bit support can be introduced in the future via a separate type BITMAP64 or extension to BITMAP.

Public Interfaces

Bitmap Interface

The Bitmap interface serves as the unified API for both user code and Flink's internal components. All modifying operations (e.g., add, and, or) are performed in-place for efficiency.

Flink provides a single built-in implementation based on RoaringBitmap. Custom implementations of this interface are not supported and will be rejected at runtime to ensure serialization safety and compatibility. Users requiring direct access to RoaringBitmap's advanced features can operate on an external RoaringBitmap instance and convert it to Flink's Bitmap via Bitmap.fromRoaring() (which performs a deep copy).

Bitmap

```
package org.apache.flink.types.bitmap;

/**
 * A compressed data structure for storing sets of 32-bit integers.
 *
 * <p>The modifying methods in this interface modify the bitmap in place
 * by default. Consider using {@link Bitmap#from(Bitmap other)} to create
 * a copied bitmap before modification if immutability is required.
 */
@PublicEvolving
public interface Bitmap {

    /** Adds the value to the bitmap. */
```

Bitmap

```
    void add(int value);

    /** Adds all integers in [rangeStart,rangeEnd) to the bitmap. */
    void add(long rangeStart, long rangeEnd);

    /**
     * Adds the first n elements of the specified array starting at the
     * specified offset.
     */
    void addN(int[] values, int offset, int n);

    /**
     * Performs an in-place logical AND (intersection) operation with another
     * bitmap.
     *
     * <p>Does nothing if {@code other} is null.
     */
    void and(@Nullable Bitmap other);

    /**
     * Performs an in-place logical AND-NOT (difference) operation with another
     * bitmap, which is equivalent to {@code this AND (NOT other)}.
     *
     * <p>Does nothing if {@code other} is null.
     */
    void andNot(@Nullable Bitmap other);

    /** Resets to an empty bitmap. */
    void clear();

    /** Checks whether the value appears in the bitmap. */
    boolean contains(int value);

    /** Gets the number of distinct values in the bitmap. */
    int getCardinality();

    /**
     * Gets the number of distinct values in the bitmap. This returns a full
     * 64-bit result.
     */
    long getLongCardinality();

    /** Checks whether the bitmap is empty. */
    boolean isEmpty();

    /**
     * Performs an in-place logical OR (union) operation with another bitmap.
     *
```

Bitmap

```
     * <p>Does nothing if {@code other} is null.
     */
    void or(@Nullable Bitmap other);

    /** Removes the value from the bitmap. */
    void remove(int value);

    /**
     * Converts the bitmap to an array of 32-bit integers, the values are sorted
     * by {@link Integer#compareUnsigned}. Avoid calling this method if the
     * bitmap is too large.
     */
    int[] toArray();

    /**
     * Converts the bitmap to an array of bytes.
     *
     * <p>Following the format defined in <a
     * href="https://github.com/RoaringBitmap/RoaringFormatSpec">32-bit
     * RoaringBitmap format specification</a>.
     */
    byte[] toBytes();

    /**
     * Converts the bitmap to a string, the values are sorted by {@link
     * Integer#compareUnsigned}. The string will be truncated and end with "..."
     * if it is too long.
     *
     * <p>For example:
     *
     * <ul>
     *   <li>{@code "{}"}, {@code "{1,2,3,4,5}"}
     *   <li>Negative values (converted to unsigned): {@code
     *   "{0,1,4294967294,4294967295}"}
     *   <li>String too long: {@code "{1,2,3,...}"}
     * </ul>
     */
    String toString();

    /**
     * Performs an in-place logical XOR (symmetric difference) operation with
     * another bitmap.
     *
     * <p>Does nothing if {@code other} is null.
     */
    void xor(@Nullable Bitmap other);

    // ~ Static Methods ----------------------------------------------------
```

Bitmap

```
    /** Gets an empty bitmap. */
    static Bitmap empty() {}

    /** Gets a copied bitmap. Returns null if {@code other} is null. */
    static Bitmap from(Bitmap other) {}

    /**
     * Gets a bitmap from an array of bytes. Returns null if {@code bytes} is
     * null.
     *
     * <p>Following the format defined in <a
     * href="https://github.com/RoaringBitmap/RoaringFormatSpec">32-bit
     * RoaringBitmap format specification</a>.
     */
    static Bitmap fromBytes(byte[] bytes) {}

    /**
     * Gets a bitmap from an array of values. Returns null if {@code values} is
     * null.
     */
    static Bitmap fromArray(int[] values) {}

    /**
     * Gets a bitmap by copying the content of the given {@link RoaringBitmap}.
     * Returns null if {@code roaringBitmap} is null.
     */
    static Bitmap fromRoaring(RoaringBitmap roaringBitmap) {}
}
```

DataStream API

Types

```
package org.apache.flink.api.common.typeinfo;

@PublicEvolving
public class Types {

    /**
     * Returns type information for {@link Bitmap}. Supports a null
     * value.
     */
    public static final TypeInformation<Bitmap> BITMAP =
BitmapTypeInfo.INSTANCE;

    ...
```

Types

```
}
```

BitmapTypeInfo

```
package org.apache.flink.api.common.typeinfo;

/** Type information for {@link Bitmap}. */
@PublicEvolving
public class BitmapTypeInfo extends TypeInformation<Bitmap> {

    public static final BitmapTypeInfo INSTANCE = new BitmapTypeInfo();

    ...
}
```

Table API / SQL

ArrayData

```
package org.apache.flink.table.data;

@PublicEvolving
public interface ArrayData {
    ...

    /** Returns the bitmap value at the given position. */
    Bitmap getBitmap(int pos);
}
```

RowData

```
package org.apache.flink.table.data;

@PublicEvolving
public interface RowData {
    ...

    /** Returns the bitmap value at the given position. */
    Bitmap getBitmap(int pos);
}
```

DataTypes

```
package org.apache.flink.table.api;

@PublicEvolving
public final class DataTypes {
    ...

   /**
     * Data type of bitmap data.
     *
     * <p>The type supports storing 32-bit integers in a compressed form.
     *
     * @see BitmapType
     */
    public static DataType BITMAP() {
        return new AtomicDataType(new BitmapType());
    }
}
```

The default conversion class of BitmapType is Bitmap.

BitmapType

```
package org.apache.flink.table.types.logical;

/**
 * Data type of bitmap data.
 *
 * <p>This type supports storing 32-bit integers in a compressed form. This can
 * be useful for efficiently representing and querying large sets of integers.
 *
 * <p>The serializable string representation of this type is {@code BITMAP}.
 */
@PublicEvolving
public final class BitmapType extends LogicalType {
    ...
}
```

LogicalTypeRoot

```
package org.apache.flink.table.types.logical;

@PublicEvolving
public enum LogicalTypeRoot {
    ...

    BITMAP(LogicalTypeFamily.EXTENSION);
```

LogicalTypeRoot
}

Built-in Functions

The initial set of built-in functions for the BITMAP type consists of 15 (4 agg, 11 scalar) functions, covering construction, logical operations, and output conversion. Additional operations such as point/range queries and iteration can be added in the future based on user feedback.

All function names are uniformly prefixed with “bitmap_”. This follows the industry standard—most systems use either “bitmap_” (ClickHouse, Doris, StarRocks) or “rb_” (some PostgreSQL extensions) as the prefix for bitmap operations.

SQL	Table API	Description
BITMAP_BUILD_AGG(value)	value.bitmapBuildAgg()	Aggregates 32-bit integers into a bitmap.
BITMAP_AND_AGG(bitmap)	bitmap.bitmapAndAgg()	Aggregates the AND (intersection) of multiple bitmaps.
BITMAP_OR_AGG(bitmap)	bitmap.bitmapOrAgg()	Aggregates the OR (union) of multiple bitmaps.
BITMAP_XOR_AGG(bitmap)	bitmap.bitmapXorAgg()	Aggregates the XOR (symmetric difference) of multiple bitmaps.
BITMAP_BUILD(array)	array.bitmapBuild()	Creates a bitmap from an array of 32-bit integers.
BITMAP_CARDINALITY(bitmap)	bitmap.bitmapCardinality()	Returns the 32-bit cardinality of a bitmap.
BITMAP_LONG_CARDINALITY(bitmap)	bitmap.bitmapLongCardinality()	Returns the 64-bit cardinality of a bitmap.
BITMAP_AND(bitmap1, bitmap2)	bitmap1.bitmapAnd(bitmap2)	Computes the AND (intersection) of two bitmaps.
BITMAP_OR(bitmap1, bitmap2)	bitmap1.bitmapOr(bitmap2)	Computes the OR (union) of two bitmaps.
BITMAP_XOR(bitmap1, bitmap2)	bitmap1.bitmapXor(bitmap2)	Computes the XOR (symmetric difference) of two bitmaps.
BITMAP_ANDNOT(bitmap1, bitmap2)	bitmap1.bitmapAndnot(bitmap2)	Computes the AND NOT (difference) of two bitmaps.

SQL	Table API	Description
BITMAP_FROM_BYTES(bytes)	bytes.bitmapFromBytes()	Converts an array of bytes to a bitmap following the standard 32-bit RoaringBitmap binary format[6].
BITMAP_TO_BYTES(bitmap)	bitmap.bitmapToBytes()	Converts a bitmap to an array of bytes following the standard 32-bit RoaringBitmap binary format.
BITMAP_TO_ARRAY(bitmap)	bitmap.bitmapToArray()	Converts a bitmap to an array of 32-bit integers, the values are sorted by Integer.compareUnsigned.
BITMAP_TO_STRING(bitmap)	bitmap.bitmapToString()	Converts a bitmap to a string, the values are sorted by Integer.compareUnsigned. The string will be truncated and end with "..." if it is too long.

Proposed Changes

Format and Conversions

Serialization Format

The BITMAP type uses the standard 32-bit RoaringBitmap binary format for serialization. This ensures:

- Full interoperability with systems like ClickHouse, StarRocks, and Doris;
- Compatibility across Flink versions;
- No Flink-specific headers or magic bytes.

The RoaringBitmap binary format was extended in library version 0.5.0 with the introduction of run containers to better compress consecutive integers. Bitmaps serialized by Flink may contain run containers and cannot be read by systems using RoaringBitmap of lower versions.

Output Conversions

Values added to Bitmap (Java int, in order)	String Bitmap#toString()	Array Bitmap#toArray()
-	“{}”	[]
4, 1, 0	“{0,1,4}”	[0,1,4]
-1, -3, 0, 2	“{0,2,4294967293,4294967295}”	[0,2,-3,-1]

Values added to Bitmap (Java int, in order)	String Bitmap#toString()	Array Bitmap#toArray()
0, 1, 2, …, 1000000	“{0,1,2<,...>,...}”	[0,1,2<,…>,1000000]

* The values are sorted by Integer.compareUnsigned.
* String output will be truncated and end with “…” if it is too long.
* <,...> denotes omitted middle elements.

Internal Implementation Details

RoaringBitmap32Data is the internal implementation of Bitmap, which wraps the widely used RoaringBitmap library. We plan to depend on RoaringBitmap v1.5.3 (the latest stable version available on GitHub[7]). Due to ongoing publishing issues in the RoaringBitmap community[7], the latest versions are not available on Maven Central. Following the official recommendation in the GitHub repository, we will use JitPack to depend on v1.5.3.

Flink guarantees that serialized bitmaps produced by its own operations are valid, and therefore does not perform additional validation during deserialization to avoid unnecessary overhead. If bitmap data originates from external systems or has been modified outside of Flink, the validation responsibility lies with those external systems.

For future version upgrades, we will adopt a conservative approach and only upgrade when new releases address issues relevant to Flink's usage scenarios.

pom.xml (flink-core)

```xml
<!-- Bitmap internal implementation -->
<dependencies>
    <dependency>
        <groupId>com.github.RoaringBitmap.RoaringBitmap</groupId>
        <artifactId>roaringbitmap</artifactId>
        <version>1.5.3</version>
    </dependency>
</dependencies>

<repositories>
    <repository>
        <id>jitpack.io</id>
        <url>https://jitpack.io</url>
    </repository>
</repositories>
```

RoaringBitmap32Data

```
package org.apache.flink.types.bitmap;
```

RoaringBitmap32Data

```
/**
 * An internal Bitmap implementation that wraps {@link RoaringBitmap} for
 * Flink-specific modifications.
 */
@Internal
public final class RoaringBitmap32Data implements Bitmap {

    private final RoaringBitmap roaringBitmap;
    ...
}
```

BitmapSerializer

```
package org.apache.flink.api.common.typeutils.base;

/** Serializer for {@link Bitmap}. */
@Internal
public class BitmapSerializer extends TypeSerializerSingleton<Bitmap> {

    public static final BitmapSerializer INSTANCE = new BitmapSerializer();
    ...
}
```

Work Plan

Phase 1: Core Functionality

1. Support BITMAP type in Calcite parser
2. Introduce Bitmap type for DataStream API
3. Introduce BITMAP type for Table API/SQL
4. Add BITMAP built-in functions
5. Add documentation and example for BITMAP type

Phase 2: Extended Ecosystem Support

6. Add more cast rules from/to BITMAP
7. Support BITMAP type in Parquet and other formats
8. Support BITMAP type for Python UDF
9. …

Performance Considerations

Large Bitmap Storage

Although the BITMAP type provides significant compression for integer sets, its actual memory footprint depends heavily on data distribution and cannot be precisely estimated. In the worst case—when storing sparse, randomly distributed integers across the full 32-bit range (excluding negative values)—a serialized bitmap may occupy up to approximately 250MB of memory.

Best Practice: BITMAP achieves optimal compression with consecutive or clustered integers. Such data patterns can be efficiently compressed by RoaringBitmap's run containers. Consider using dictionary encoding or bucketing strategies to transform your data into consecutive ranges when possible.

State Access Overhead

Since BITMAP is stored as a single serialized object in Flink's state backend, every state access requires full deserialization, and every update requires full serialization. For large bitmaps, this serialization overhead can become noticeable in high-throughput streaming aggregations.

Mitigation Strategies:

- Enable mini-batch aggregation: Reduce state access frequency by buffering multiple records before updating state;
- Use multi-level aggregation: Pre-aggregate at finer granularity (e.g., per minute) before computing coarser aggregations (e.g., per hour), as demonstrated in the example.

Example

Per-Minute UV with Bitmap Storage for Flexible Analysis

```
CREATE TABLE user_events (
  user_id INT,
  tag STRING,
  event_time TIMESTAMP(3),
  WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
  'connector' = 'kafka',
  ...
);

CREATE TABLE minute_bitmaps (
  window_start TIMESTAMP(3),
  tag STRING,
```

Per-Minute UV with Bitmap Storage for Flexible Analysis

```
  user_bitmap BYTES,
  WATERMARK FOR window_start AS window_start - INTERVAL '1' MINUTE
) WITH (
  'connector' = 'jdbc',
  ...
);

CREATE TABLE hourly_common_uv (
  hour_start TIMESTAMP(3),
  common_uv INT
) WITH (
  'connector' = 'jdbc',
  ...
);

EXECUTE STATEMENT SET
BEGIN

INSERT INTO minute_bitmaps
SELECT
  window_start,
  tag,
  BITMAP_TO_BYTES(BITMAP_BUILD_AGG(user_id)) AS user_bitmap
FROM TABLE(
  TUMBLE(TABLE user_events, DESCRIPTOR(event_time), INTERVAL '1' MINUTE)
)
GROUP BY window_start, tag;

INSERT INTO hourly_common_uv
SELECT
  window_start AS hour_start,
  BITMAP_CARDINALITY(
    BITMAP_AND(
      BITMAP_OR_AGG(BITMAP_FROM_BYTES(user_bitmap)) FILTER (WHERE tag = 'A'),
      BITMAP_OR_AGG(BITMAP_FROM_BYTES(user_bitmap)) FILTER (WHERE tag = 'B')
    )
  ) AS common_uv
FROM TABLE(
  TUMBLE(TABLE minute_bitmaps, DESCRIPTOR(window_start), INTERVAL '1' HOUR)
)
GROUP BY window_start;

END;
```

Compatibility, Deprecation, and Migration Plan

This FLIP integrates a new data type in Flink, and it is fully backward compatible.

Test Plan

The change will be covered with unit and integration tests.

Rejected Alternatives

None.

Future Works

Flink-specific RoaringBitmap

To enable deeper customization, we plan to integrate a modified version of the RoaringBitmap source code directly into Flink's codebase (under appropriate licensing). This would allow us to:

- Perform low-level operations on internal containers, such as direct access, partial updates, or custom container types, which are not exposed through the current black-box wrapper;
- Support efficient incremental processing for COUNT(DISTINCT) aggregations;
- Evolve the implementation independently while maintaining compatibility with the standard Roaring serialization format for interoperability with external systems.

Optimized State Access

In current bitmap-based aggregations, such as BITMAP_BUILD_AGG, the entire bitmap is deserialized from state on every access, even for operations that only affect a small portion of the data. Since the RoaringBitmap is composed of independent containers (each ≤ 8 KB), we plan to explore storing it as a sharded structure like MapState<Key, Container>. This would enable:

- Lazy, partial deserialization: Only the relevant containers are loaded during runtime operations;
- Reduced CPU and memory pressure during stateful processing, especially for large bitmaps;
- Faster per-record processing in aggregation functions, as most operations touch only a few containers.

Extended Query and Iteration Capabilities

The initial BITMAP implementation focuses on core aggregation and set operations. Future enhancements may include:

- Point queries: BITMAP_CONTAINS() for point lookups;

- Range queries: BITMAP_MIN(), BITMAP_MAX(), BITMAP_RANGE();
- Iterator support: Efficient streaming iteration over bitmap elements without full materialization via toArray();
- Batch operations: BITMAP_REMOVE_RANGE(), BITMAP_FLIP() for bulk modifications;
- Advanced predicates: BITMAP_INTERSECTS() for fast intersection checks without computing the full result.

These extensions would enable more flexible bitmap manipulation and complex analytical queries.

Reference

[1] https://roaringbitmap.org/
[2] https://clickhouse.com/docs/sql-reference/functions/bitmap-functions
[3] https://docs.starrocks.io/docs/sql-reference/data-types/other-data-types/BITMAP/
[4] https://doris.apache.org/docs/3.x/sql-manual/basic-element/sql-data-types/aggregate/BITMAP
[5] https://github.com/ChenHuajun/pg_roaringbitmap
[6] https://github.com/RoaringBitmap/RoaringFormatSpec
[7] https://github.com/RoaringBitmap/RoaringBitmap
[8] https://github.com/RoaringBitmap/RoaringBitmap/issues/749