# FLIP-37: Rework of the Table API Type System (Part 1)

Author: Timo Walther, Kurt Young, Dawid Wysakowicz, Xuefu Zhang, Jingsong Li

Last update: 2019-05-09

**Disclaimer:**
**This document is a draft for the long-term vision!**
**The exact implementation plan needs to be determined!**

## Motivation

Currently, the Table & SQL API relies on Flink's TypeInformation at different positions in the code base. The API uses it for conversion between DataSet/DataStream API, casting, and table schema representation. The planning for code generation and serialization of runtime operators.

The past has shown that TypeInformation is useful when converting between DataSet/DataStream API, however, it does not integrate nicely with SQLs type system and depends on the programming language that is used.

For example, if users have implemented a TableFunction:

```
case class SimpleUser(name: String, age: Int)

class TableFunc0 extends TableFunction[SimpleUser] {
  // make sure input element's format is "<string>#<int>"
  def eval(user: String): Unit = {
    if (user.contains("#")) {
      val splits = user.split("#")
      collect(SimpleUser(splits(0), splits(1).toInt))
    }
  }
}
```

The return type of this table function does not only depend on the function class itself but also on the table environment that is used:

```
org.apache.flink.table.api.java.StreamTableEnvironment#registerFunction
```

Uses the Java type extraction stack and extracts TypeInformation by using the reflection-based TypeExtractor.

`org.apache.flink.table.api.scala.StreamTableEnvironment#registerFunction`

Uses the Scala type extraction stack and extracts TypeInformation by using a Scala macro.

Depending on the table environment, the example above might be serialized using a Case Class serializer or a Kryo serializer (I assume the case class is not recognized as a POJO).

The inflexibility and inconsistency has also been mentioned by other big contributors such as Uber. See:

[FLINK-9484] Improve generic type inference for User-Defined Functions
[FLINK-9294] Improve type inference for UDFs with composite parameter or result type
[FLINK-9501] Allow Object or Wildcard type in user-define functions as parameter types but not result types
[FLINK-9502] Use generic parameter search for user-define functions when argument contains Object type
[FLINK-9430] Support Casting of Object to Primitive types for Flink SQL UDF

The current type system has many different shortcomings.

- It is not aligned with SQL.
- For example, precision and scale can not be defined for DECIMALs.
- The difference between CHAR/VARCHAR is not supported (FLINK-10257, FLINK-9559).
- Physical type and logical type are tightly coupled.
- Physical type is type information instead of type serializer.

## Goals

This document proposes a complete rework of the Table & SQL API type system. For making the API stable and be comparable or better than existing SQL processors.

This document helps in aligning API, planners, and connectors/formats/catalogs.

Part 2 will discuss how to integrate the type system with UDFs and expressions.

# Background

Some research how other SQL vendors dealing with this problem and what the SQL standard says.

## Standard SQL's Data Type

### Families

PREDEFINED - according to SQL standard
CONSTRUCTED - according to SQL standard
USER_DEFINED - according to SQL standard
CHARACTER_STRING - according to SQL standard
BINARY_STRING - according to SQL standard
LARGE_OBJECT_STRING - according to SQL standard
EXACT_NUMERIC - according to SQL standard
APPROXIMATE_NUMERIC - according to SQL standard
NUMERIC - according to SQL standard
TIME - according to SQL standard
TIMESTAMP - according to SQL standard
DATETIME - according to SQL standard
INTERVAL - according to SQL standard
COLLECTION - according to SQL standard
DISTINCT - according to SQL standard
STRUCTURED - according to SQL standard

### Data Types

| **SQL standard** | **Families** |
|---|---|
| CHAR | PREDEFINED<br>CHARACTER_STRING |
| VARCHAR | PREDEFINED<br>CHARACTER_STRING |
| CLOB | PREDEFINED<br>CHARACTER_STRING<br>LARGE_OBJECT_STRING |
| BINARY | PREDEFINED<br>BINARY_STRING |

| | |
|---|---|
| VARBINARY | PREDEFINED<br>BINARY_STRING |
| BLOB | PREDEFINED<br>BINARY_STRING<br>LARGE_OBJECT_STRING |
| NUMERIC | PREDEFINED<br>EXACT_NUMERIC<br>NUMERIC |
| DECIMAL | PREDEFINED<br>EXACT_NUMERIC<br>NUMERIC |
| SMALLINT | PREDEFINED<br>EXACT_NUMERIC<br>NUMERIC |
| INT | PREDEFINED<br>EXACT_NUMERIC<br>NUMERIC |
| BIGINT | PREDEFINED<br>EXACT_NUMERIC<br>NUMERIC |
| FLOAT | PREDEFINED<br>APPROXIMATE_NUMERIC<br>NUMERIC |
| REAL | PREDEFINED<br>APPROXIMATE_NUMERIC<br>NUMERIC |
| DOUBLE PRECISION | PREDEFINED<br>APPROXIMATE_NUMERIC<br>NUMERIC |
| DECFLOAT | PREDEFINED<br>NUMERIC |
| BOOLEAN | PREDEFINED |
| DATE | PREDEFINED<br>DATETIME |
| TIME [WITHOUT | PREDEFINED<br>TIME |

| | |
|---|---|
| TIMEZONE] | DATETIME |
| TIME [WITH TIMEZONE] | PREDEFINED<br>TIME<br>DATETIME |
| TIMESTAMP [WITHOUT TIMEZONE] | PREDEFINED<br>TIMESTAMP<br>DATETIME |
| TIMESTAMP [WITH TIMEZONE] | PREDEFINED<br>TIMESTAMP<br>DATETIME |
| INTERVAL | PREDEFINED<br>INTERVAL |
| ARRAY | CONSTRUCTED<br>COLLECTION |
| MULTISET | CONSTRUCTED<br>COLLECTION |
| REF | CONSTRUCTED |
| ROW | CONSTRUCTED |

## Hive's Data Types

Source:
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types
https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-TypeSystem

| **Type** | **Comment** |
|---|---|
| TINYINT | 1-byte signed integer, from -128 to 127 |
| SMALLINT | 2-byte signed integer, from -32,768 to 32,767 |
| INT/INTEGER | 4-byte signed integer, from -2,147,483,648 to 2,147,483,647 |
| BIGINT | 8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| FLOAT | 4-byte single precision floating point number |

| | |
|---|---|
| DOUBLE | 8-byte double precision floating point number |
| DOUBLE PRECISION | alias for DOUBLE |
| DECIMAL | user-definable precision and scale;<br>DECIMAL, -- Defaults to decimal(10,0) |
| NUMERIC | same as DECIMAL |
| TIMESTAMP | Supports traditional UNIX timestamp with optional nanosecond precision.<br>Supported conversions:<br><br>Integer numeric types: Interpreted as UNIX timestamp in seconds<br><br>Floating point numeric types: Interpreted as UNIX timestamp in seconds with decimal precision<br><br>Strings: JDBC compliant java.sql.Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)<br><br>Support for TIMESTAMP WITH LOCAL TIME ZONE is aimed. |
| DATE | particular year/month/day, in the form YYYY-MM-DD |
| INTERVALS | 2 separate interval types:<br>year-month interval<br>day-time interval |
| STRING | unbounded |
| VARCHAR | between 1 and 65535; If a string value being converted/assigned to a varchar value exceeds the length specifier, the string is silently truncated; Character length is determined by the number of code points contained by the character string; Non-generic UDFs cannot directly use varchar type as input arguments or return values. |
| CHAR | Fixed-length; maximum length is fixed at 255 |
| BOOLEAN | |
| BINARY | Byte array |
| ARRAY<data_type> | |
| MAP<primitive_type, data_type> | |
| STRUCT<col_name : data_type [COMMENT col_comment], ...> | |
| UNIONTYPE<data_type, | full support for this type in Hive remains incomplete |

| | |
|---|---|
| data_type, ...> | |

## Blink's Data Types

| Table API | Class | Comment |
|---|---|---|
| STRING | StringType | Externally String, internally UTF-8 binary string |
| BOOLEAN | BooleanType | |
| DOUBLE | DoubleType | |
| FLOAT | FloatType | |
| BYTE | ByteType | |
| INT | IntType | |
| LONG | LongType | |
| SHORT | ShortType | |
| CHAR | CharType | |
| BYTE_ARRAY | ByteArrayType | |
| DATE<br>INTERVAL_MONTHS | DateType(id, name) | Externally java.sql.Date, internally int<br><br>With ID:<br>0 = DateType<br>1 = IntervalMonths |
| TIMESTAMP<br>INTERVAL_MILLIS<br>ROWTIME_INDICATOR<br>PROCTIME_INDICATOR | TimestampType(id, name) | Externally java.sql.Timestamp, internally long<br><br>With ID:<br>0 = TimestampType<br>1 = IntervalMillis<br>2 = RowTimeIndicator<br>3 = ProctimeTimeIndicator |
| TIME | TimeType | Externally java.sql.Time, internally int |
| INTERVAL_ROWS | IntervalRowsType | |
| INTERVAL_RANGE | IntervalRangeType | Result type for OVER windows |
| createArrayType()<br>createPrimitiveArrayType() | ArrayType | isPrimitive, elementType |

| | | |
|---|---|---|
| createDecimalType | DecimalType(precision, scale) | Default to be same with Integer<br>USER_DEFAULT = new DecimalType(10, 0)<br>SYSTEM_DEFAULT = new DecimalType(38, 18) |
| createMapType | MapType | |
| createGenericType | GenericType(typeInfo) | No execution config considered for serializers. |
| createMultisetType | MultisetType extends MapType | |
| createRowType | RowType | Externally Row, internally BaseRow |
| createTupleType<br>pojoBuilder<br>extractDataType | TypeInfoWrappedDataType | External type that is mapped to an internal type above.<br>E.g. row/pojo/caseclass to row type |

## Other resources

https://www.cockroachlabs.com/docs/stable/data-types.html
https://www.ibm.com/support/knowledgecenter/en/SSGU8G_12.1.0/com.ibm.sqlr.doc/ids_sqr_094.htm
https://crate.io/docs/sql-99/en/latest/index.html

# Flink Type System Concepts

We propose the following type system concepts.

## Simplifications

- **Encoding**
  - Flink's Table & SQL API will not support different encodings.
  - Internally, string types in Flink are always UTF-8 encoded.
  - Connectors and formats that return Java `String`s might produce UTF-16 but they are converted immediately as part of the "source conversion step".
  - The serialization and deserialization of other encodings is the responsibility of a format/connector. Thus, connectors that work on binary data must ensure UTF-8 input/output.
  - The same principles apply to user-defined functions.
- **Collation**
  - Flink's Table & SQL API will not support different collations.
  - Internally, string types are always compared using UTF-8 code point comparison.

## Concept

Flink distinguishes between DataType and LogicalType. DataType is exposed to the user and immediately converted into a LogicalType internally for planning and validation.

- **DataType<T>**
  - User-facing data type representation for both Table API and SPI classes such as table sources or table sinks.
  - Wraps a logical type and adds additional information about physical representation and how to convert it at planner/runtime boundaries.
  - DataType is parameterized by a Class T.
  - Uncouples from Flink's TypeInformation.
  - DataType is similar to Blink's ExternalType.
  - Properties:
    - Immutable
    - Java Serializable
  - Contains:
    - a LogicalType for logical representation
      (e.g. TIMESTAMP(3))

        - a Class T for physical representation hints for the runtime. It allows users to specify what is expected in a connector or UDF.
          (e.g. Long, java.sql.Timestamp, java.time.LocalDateTime)
        - children subtypes matching the children of the logical type
    - Interface:
        - getLogicalType(): LogicalType
        - getBridgingClass(): Class<T>
        - bridgeTo(Class<B>): DataType<B>
          Note: Creates a new DataType instance with a different class but same logical type. The method only allows valid conversions defined by the LogicalTypeRoot. If the class is not supported, an exception is thrown.
        - notNull(): DataType
          Note: Creates a new DataType instance with a different logical type but same physical class.
        - accept(DataTypeVisitor<R>): <R>
        - getChildren(): List<DataType>
          Allows iterating through subtypes. Either the element type of arrays, the key/value types of maps, or the fields of rows.
    - Additional notes:
        - Both DataType and user-defined types are bound to some class. The class of a UDT also defines runtime logic for methods. The DataType class can differ from the UDT class. For example, a POJO could also be represented as a Row at the boundaries of the API. The runtime calls the UDT class for methods but converts to DataType class if required.

- **LogicalType**
    - Logical, unified type close to the SQL standard.
    - Uncouples the type system from runtime specifics.
    - Uncouples from Flink's TypeInformation and Calcite's RelDataType.
    - Shares many concepts with Calcite's RelDataType.
    - Similar to Blink's InternalType.
    - Properties:
        - Immutable
        - Java Serializable
        - String Serializable
    - Interface:
        - isNullable(): Boolean
          Note: Don't confuse nullability with a SQL column list constraint, having this attribute here allows having more fine-grained internal operations when dealing with types.
        - copy(isNullable: boolean): LogicalType
        - getRoot(): LogicalTypeRoot
        - asSummaryString(): String
          Note: Intended for printing to a console.

        - asSerializableString(): String
          Note: Fully serializes the type (e.g. ANY type might be a long base64 string).
        - accept(LogicalTypeVisitor<R>): <R>
        - getChildren(): List<LogicalType>
          Allows iterating through subtypes. Either the element type of arrays, the key/value types of maps, or the fields of rows.
        - supportsBridgingTo(Class<?>): Boolean
          Note: Each LogicalType validates according to a set of supported conversion classes. Since the LogicalType is part of the flink-table-common module the validation might happen only on class names as internal data format classes cannot be referenced from within this Maven module.
          For example: VARCHAR -> String, byte[], BinaryString
        - getDefaultConversion(): Class<?>
          Each LogicalTypeRoot maps to a default conversion class.
          For example: VARCHAR -> String
- **LogicalTypeRoot**
    - Essential description of a data type without additional parameters.
    - For example:
        - VARCHAR(200) -> VARCHAR
        - ARRAY<ROW, INT> -> ARRAY
    - Defined as an enum for efficient evaluation.
    - Especially useful for comparisons.
    - The LogicalTypeRoot basically contains all static information about logical data types encoded in the table of the following chapter.
    - Each LogicalTypeRoot maps to a set of `LogicalTypeFamily`.
        - VARCHAR -> PREDEFINED, CHARACTER_STRING
- **LogicalTypeFamily**
    - Allows for clustering of data types according to common properties.
    - Defined as an enum for efficient evaluation.
    - Especially useful for validation in Table API.
    - Avoids complex class hierarchies of questionable categorization.
        - Flink's "atomic" interface mixes comparability, keyability, etc.
        - Blink's "atomic" interface includes byte arrays and contains no methods.
- **DataTypes**
    - Lists all available logical types wrapped into DataType with default conversion.
    - Thus, API users must not care about conversions.
      For example:
        - DataTypes.VARCHAR(200)
        - DataTypes.TIMESTAMP(3)
    - SPI implementations can specify different conversions or optimizer hints.
      For example:

- DataTypes.VARCHAR(200).bridgeTo(String.class)
- DataTypes.ARRAY(DataTypes.BIGINT).bridgeTo(Long[].class)
- DataTypes.ARRAY(DataTypes.BIGINT.notNull()).bridgeTo(long[].class)
- DataTypes.TIMESTAMP(3).notNull().bridgeTo(long.class)

# Flink Logical Type Proposal

We propose the following logical types.

## Families

PREDEFINED - according to SQL standard
CONSTRUCTED - according to SQL standard
USER_DEFINED - according to SQL standard
CHARACTER_STRING - according to SQL standard
BINARY_STRING - according to SQL standard
EXACT_NUMERIC - according to SQL standard
APPROXIMATE_NUMERIC - according to SQL standard
NUMERIC - according to SQL standard
TIMESTAMP - according to SQL standard
DATETIME - according to SQL standard
INTERVAL - according to SQL standard
COLLECTION - according to SQL standard

EXTENSION - Extension to the SQL standard

## Logical Types

**General notes**:

- Hive compatibility: Hive compatibility is not required in Flink SQL's main API. Instead, standard compliance should be aimed. Hive compatibility should be handled by an additional compatibility layer on top. However, Flink needs to provide a well defined set of data types for smooth mapping and integration.
- String lengths: The SQL standard defines minimum and default lengths but not maximum lengths. We aim for the most restrictive approach that might be relaxed in the future.
- Decimal considerations: We use Hive's maximums: https://github.com/apache/hive/blob/master/storage-api/src/java/org/apache/hadoop/hive/common/type/HiveDecimal.java
- Constructed types: We use Hive's representation for types that consist of other types because it allows for modern, more complex types with arbitrary deep nesting. We use "<>" for that.
- Row interval types: A row interval is always a count and thus a regular integer. Let's keep it simple and handle that with predefined SQL types (INT).
- Time types:

- https://docs.google.com/document/d/1gNRww9mZJcHvUDCXklzjFEQGpefsuR_akCDfWsdE35Q/edit#
- https://docs.google.com/document/d/1E-7miCh4qK6Mg54b-Dh5VOyhGX8V4xdMXKIHJL36a9U/edit#heading=h.n699ftkvhjlo
- https://docs.oracle.com/cd/E11882_01/server.112/e10729/ch4datetime.htm#NLSPG004
- https://stackoverflow.com/questions/42766674/java-convert-java-time-instant-to-java-sql-timestamp-without-zone-offset
- → we will not support conversions from/to TIMESTAMP WITHOUT TIME ZONE to long/int but provide UTC functions instead, also for computed columns for time attributes
- → we will provide conversion to the legacy java.sql.* classes for backwards compatibility and JDBC support
- → we will use OffsetDateTime for timezoned timestamp in accordance to the general big data ecosystem
- → we will add a TIMESTAMP WITH LOCAL TIMEZONE type for supporting Java Instant semantics and interpretation of data that does not store timezone information in the physical values (supporters of such a type are Calcite, Oracle, Snowflake, Hive).

| **SQL** | **Table API** | **Conversion** | **Family** | **Comment** | **Flink** |
|---|---|---|---|---|---|
| CHAR(n) | CHAR(n) | In:<br>java.lang.String<br>byte[]<br>org.apache.flink.table.dataformat.BinaryString<br><br>Out:<br>java.lang.String (Default)<br>byte[]<br>org.apache.flink.table.dataformat.BinaryString | PREDEFINED<br><br>CHARACTER_STRING | n is the fixed-length number of code points.<br><br>1 <= n <= 255<br><br>Default n: 1 | 1.9+ |
| VARCHAR(n) | VARCHAR(n) | In:<br>java.lang.String<br>byte[]<br>org.apache.flink.table.dataformat.BinaryString<br><br>Out:<br>java.lang.String (Default)<br>byte[]<br>org.apache.flink.table.dataformat.BinaryString | PREDEFINED<br><br>CHARACTER_STRING | n is the variable-length number of code points.<br><br>1 <= n <= INT_MAX<br><br>Default n: 1 | 1.9+ |
| STRING | | | | For compatibility and usability: | 1.9 |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
|  |  |  |  | VARCHAR(INT_MAX) |  |
| BOOLEAN | BOOLEAN | In:<br>java.lang.Boolean<br>boolean<br><br>Out:<br>java.lang.Boolean (Default)<br>boolean | PREDEFINED | For true, false, unknown semantics. | 1.9 |
| BINARY(n) | BINARY(n) | In:<br>byte[]<br>org.apache.flink.table.dataformat.BinaryArray<br><br>Out:<br>byte[] (Default)<br>org.apache.flink.table.dataformat.BinaryArray | PREDEFINED<br><br>BINARY_STRING | n is fixed-length number of bytes.<br><br>1 <= n <= INT_MAX<br><br>Default n: 1 | 1.9+ |
| VARBINARY(n) | VARBINARY(n) | In:<br>byte[]<br>org.apache.flink.table.dataformat.BinaryArray<br><br>Out:<br>byte[] (Default)<br>org.apache.flink.table.dataformat.BinaryArray | PREDEFINED<br><br>BINARY_STRING | n is the variable-length number of bytes.<br><br>1 <= n <= INT_MAX<br><br>Default n: 1 | 1.9 |
| BYTES |  |  |  | For compatibility and usability: VARBINARY(INT_MAX) | 1.9 |
| DECIMAL(p, s) | DECIMAL(p,s) | In:<br>java.math.BigDecimal<br>org.apache.flink.table.dataformat.Decimal<br><br>Out:<br>java.math.BigDecimal (Default)<br>org.apache.flink.table.dataformat.Decimal | PREDEFINED<br><br>EXACT_NUMERIC<br><br>NUMERIC | p = precision<br>s = scale (optional)<br><br>1 <= p <= 38<br>0 <= s <= p<br><br>Default p: 10<br>Default s: 0<br><br>Limited Flink compatibility because different defaults. | 1.9 |
| NUMERIC |  |  |  | For compatibility: DECIMAL(p, s) | 1.9 |
| TINYINT | TINYINT | In:<br>Byte<br>byte | PREDEFINED<br><br>EXACT_NUMERI | 1-byte signed integer, from -128 to 127<br><br>Primitive output type depends | 1.9 |

| | | | | | |
|---|---|---|---|---|---|
| | | Out:<br>Byte (Default)<br>byte | C<br><br>NUMERIC | on nullability. | |
| SMALLINT | SMALLINT | In:<br>Short<br>short<br><br>Out:<br>Short (Default)<br>short | PREDEFINED<br><br>EXACT_NUMERIC<br>NUMERIC | 2-byte signed integer, from -32,768 to 32,767<br><br>Primitive output type depends on nullability. | 1.9 |
| INT | INT | In:<br>Integer<br>int<br><br>Out:<br>Integer (Default)<br>int | PREDEFINED<br><br>EXACT_NUMERIC<br>NUMERIC | 4-byte signed integer, from -2,147,483,648 to 2,147,483,647<br><br>Primitive output type depends on nullability. | 1.9 |
| INTEGER | | | | For compatibility:<br>INT | 1.9 |
| BIGINT | BIGINT | In:<br>Long<br>long<br><br>Out:<br>Long (Default)<br>long | PREDEFINED<br><br>EXACT_NUMERIC<br><br>NUMERIC | 8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807<br><br>Primitive output type depends on nullability. | 1.9 |
| FLOAT | FLOAT | In:<br>Float<br>float<br><br>Out:<br>Float (Default)<br>float | PREDEFINED<br><br>APPROXIMATE_NUMERIC<br><br>NUMERIC | 4-byte single precision floating point number<br><br>Primitive output type depends on nullability. | 1.9 |
| DOUBLE | DOUBLE | In:<br>Double<br>double<br><br>Out:<br>Double (Default)<br>double | PREDEFINED<br><br>APPROXIMATE_NUMERIC<br><br>NUMERIC | 8-byte double precision floating point number<br><br>Primitive output type depends on nullability. | 1.9 |
| DOUBLE PRECISION | | | | For compatibility:<br>DOUBLE | 1.9 |

| | | | | | |
|---|---|---|---|---|---|
| DATE | DATE | In:<br>java.sql.Date<br>java.time.LocalDate<br>int<br><br>Out:<br>java.sql.Date<br>java.time.LocalDate (Default)<br>int | PREDEFINED<br><br>DATETIME | Range:<br>0000-01-01 to 9999-12-31<br><br>Limited Flink compatibility because LocalDate becomes the default.<br><br>int = number of days since epoch<br><br>Primitive output type depends on nullability. | 1.9 |
| TIME(p) [WITHOUT TIMEZONE] | TIME(p) | In:<br>java.sql.Time<br>java.time.LocalTime<br>int<br>long<br><br>Out:<br>java.sql.Time<br>java.time.LocalTime (Default)<br>int<br>long | PREDEFINED<br><br>DATETIME | Nanosecond precision.<br><br>0 <= p <= 9<br><br>Default p: 0<br><br>Limited Flink compatibility because LocalTime becomes the default and introduces a precision.<br><br>int = number of millis of day<br>long = number of nanos of day<br><br>Primitive output type depends on nullability. | 1.9 |
| TIMESTAMP(p) [WITHOUT TIMEZONE] | TIMESTAMP(p) | In:<br>java.sql.Timestamp<br>java.time.LocalDateTime<br><binary format?><br><br>Out:<br>java.sql.Timestamp<br>java.time.LocalDateTime (Default)<br><binary format?> | PREDEFINED<br><br>TIMESTAMP<br><br>DATETIME | Nanosecond precision.<br><br>String representation:<br>“YYYY-MM-DD HH:MM:SS.fffffffff”<br><br>0 <= p <= 9<br><br>Default p: 6<br><br>Limited Flink compatibility because LocalDateTime becomes the default and we introduce a precision.<br><br>Internally, the timestamp type can be marked as one of:<br>- REGULAR (Default)<br>- ROWTIME<br>- PROCTIME<br><br>The timestamp types are not | 1.9 |

| | | | | | |
|---|---|---|---|---|---|
| | | | | exposed through API. | |
| TIMESTAMP(p) WITH TIMEZONE | TIMESTAMP_WITH_TIMEZONE(p) | In:<br>java.time.OffsetDateTime<br>java.time.ZonedDateTime<br><binary format?><br><br>Out:<br>java.time.OffsetDateTime (Default)<br><binary format?> | PREDEFINED<br><br>TIMESTAMP<br><br>DATETIME | Nanosecond precision.<br><br>String representation: “YYYY-MM-DD HH:MM:SS.fffffffff-/+oo:oo”<br><br>0 <= p <= 9<br><br>Default p: 6<br><br>Internally, the timestamp type can be marked as one of:<br>- REGULAR (Default)<br>- ROWTIME<br>- PROCTIME<br><br>The timestamp types are not exposed through API. | 1.9+ |
| TIMESTAMP(p) WITH LOCAL TIMEZONE | TIMESTAMP_WITH_LOCAL_TIMEZONE(p) | In:<br>java.time.Instant<br>int<br>long<br><binary format?><br><br>Out:<br>java.time.Instant (Default)<br>int<br>long<br><binary format?> | PREDEFINED<br><br>TIMESTAMP<br><br>DATETIME<br><br>EXTENSION | Nanosecond precision.<br><br>0 <= p <= 9<br><br>Default p: 6<br><br>Similar to a TIMESTAMP WITH TIMEZONE but without time zone information stored in the physical data.<br><br>Semantically equivalent to Java's Instant.<br><br>Enables interpretation of UTC timestamps of formats/connectors into the timezone of the environment (=TableConfig.getZoneId()).<br><br>int = number of seconds since epoch<br>long = number of millis since epoch<br><br>Internally, the timestamp type can be marked as one of:<br>- REGULAR (Default)<br>- ROWTIME<br>- PROCTIME<br><br>The timestamp types are not | 1.9? |

| | | | | | |
|---|---|---|---|---|---|
| | | | | exposed through API.<br><br>Primitive output type depends on nullability. | |
| INTERVAL YEAR(p)<br><br>INTERVAL YEAR(p) TO MONTH<br><br>INTERVAL MONTH | Examples:<br><br>INTERVAL(YEAR(n), MONTH)<br><br>INTERVAL(MONTH) | In:<br>java.time.Period<br>int<br><br>Out:<br>java.time.Period (Default)<br>int | PREDEFINED<br><br>INTERVAL | Interval of months.<br><br>1 <= p <= 4<br><br>Default p: 2<br><br>For example:<br>INTERVAL '50' MONTH<br>→ `+04-02`<br><br>int = number of months<br><br>Internally represented as int (months)?<br><br>Primitive output type depends on nullability. | 1.9 |
| INTERVAL DAY(p1)<br><br>INTERVAL DAY(p1) TO HOUR<br><br>INTERVAL DAY(p1) TO MINUTE<br><br>INTERVAL DAY(p1) TO SECOND(p2)<br><br>INTERVAL HOUR<br><br>INTERVAL HOUR TO MINUTE<br><br>INTERVAL HOUR TO SECOND(p2)<br><br>INTERVAL MINUTE<br><br>INTERVAL | Examples:<br><br>INTERVAL(MINUTE) | In:<br>java.time.Duration<br>long<br><br>Out:<br>java.time.Duration (Default)<br>long | PREDEFINED<br><br>INTERVAL | Interval of nanosecond precision.<br><br>1 <= p1 <= 6<br><br>0 <= p2 <= 9<br><br>Default p1: 2<br><br>Default p2: 6<br><br>long = number of millis<br><br>Internally represented as long (seconds) and int (nanoseconds)? Or just long (milliseconds)?<br><br>Primitive output type depends on nullability. | 1.9 |

| | | | | | |
|---|---|---|---|---|---|
| MINUTE TO SECOND(p2)<br><br>INTERVAL SECOND(p2) | | | | | |
| ARRAY<LogicalType> | ARRAY(DataType) | In:<br>DataType[]<br>org.apache.flink.table.dataformat.BinaryArray<br><br>Out:<br>DataType[] (Default)<br>org.apache.flink.table.dataformat.BinaryArray | CONSTRUCTED<br><br>COLLECTION | Both primitive (e.g. byte[]) and object (e.g. Byte[]) arrays are supported as input.<br><br>An output is always an object array by default. But primitive arrays are supported if defined with nullability: ARRAY<INT NOT NULL><br><br>Any data type can be stored in an array. | 1.9 |
| LogicalType ARRAY | | | | For SQL standard compatibility: ARRAY<LogicalType> | 1.9 |
| MULTISET<LogicalType> | MULTISET(DataType) | In:<br>java.util.Map<DataType, Integer><br>org.apache.flink.table.dataformat.BinaryMap<br><br>Out:<br>java.util.Map<DataType, Integer><br>org.apache.flink.table.dataformat.BinaryMap | CONSTRUCTED<br><br>COLLECTION | Maps an arbitrary key to an integer value.<br><br>Null values in keys ARE supported. This is different to Flink's current implementation. | 1.9 |
| LogicalType MULTISET | | | | For SQL standard compatibility: MULTISET<DataType> | 1.9 |
| MAP<LogicalType, LogicalType> | MAP(DataType, DataType) | In:<br>java.util.Map<DataType, DataType><br>org.apache.flink.table.dataformat.BinaryMap<br><br>Out:<br>java.util.Map<DataType, DataType> (Default)<br>org.apache.flink.table.dataformat.BinaryMap | CONSTRUCTED<br><br>EXTENSION | Map of key to value.<br><br>Null values in keys ARE supported. This is different to Flink's current implementation. | 1.9 |
| ROW<name LogicalType [ | ROW(names, datatypes, | In:<br>org.apache.flink.types.R | CONSTRUCTED | Generic structured type. | 1.9 |

| | | | | | |
|---|---|---|---|---|---|
| ‘string’],...> | comments)<br><br>+ a builder pattern | ow<br>org.apache.flink.types.Structured<br>org.apache.flink.table.dataformat.BaseRow<br><br>Out:<br>org.apache.flink.types.Row (Default)<br>org.apache.flink.types.Structured<br>org.apache.flink.table.dataformat.BaseRow | | For future extensibility and lazy deserialization, we introduce a new interface \`Structured\` with the base interface of \`Row\`. Row will extend \`Structured\`. | |
| ROW(name LogicalType,...) | | | | For SQL standard compatibility:<br>ROW(name DataType,...)<br><br>Comments are not allowed in SQL standard. | 1.9 |
| NULL | NULL | In:<br>java.lang.Object<br><br>Out:<br>any non-primitive class<br>java.lang.Object (Default) | EXTENSION | Logical helper type for representing untyped null literals. This is required both in API calls and format type mapping.<br><br>For example, both Avro and JSON define such a NULL type.<br><br>A value except for null should not exist in any physical type as it is always null.<br><br>A NULL type can be cast to any nullable type similar to Java semantics. | 1.9+ |
| ANY(t)<br><br>ANY | ANY(t)<br><br>ANY | In:<br>Class of t<br>byte[]<br>org.apache.flink.table.dataformat.BinaryGeneric<br><br>Out:<br>Class of t (Default)<br>byte[]<br>org.apache.flink.table.dataformat.BinaryGeneric | EXTENSION | Bridging type between the SQL world and the Java world.<br><br>t = optional type information<br><br>Default t: No type information; which is equivalent to \`GenericTypeInfo(java.lang.Object)\`.<br><br>An ANY type is a black-box for SQL API and planner. | 1.9 |

| | | | | | |
|---|---|---|---|---|---|
| | | | | A plain `ANY` type is also a logical helper type for expressing that a UDF accepts and returns any kind of data type. | |
| User-defined Type | | | USER_DEFINED<br><br>STRUCTURED or DISTINCT | See next section. | 1.9? |

## User-Defined Types

**General notes:**

- SQL fully specifies user-defined types designed to work nicely with Java semantics.
- Resources:
  - https://crate.io/docs/sql-99/en/latest/chapters/27.html
  - Nice summary in this SIGMOD paper: http://synthesis.ipi.ac.ru/synthesis/student/oodb/essayRef/SQL1999.pdf
- The requirement for a CREATE TYPE DDL has been mentioned by several Flink users.
- Support for object-oriented types in SQL integrates nicely with Java POJOs, and case classes.

Note: An exact DDL is future work, here we just define the minimal set of characteristics for a unified type system between Java/Scala and SQL.

- **UserDefinedType**
  - Extends LogicalType
  - Describes a named type in SQL.
  - Characteristics:
    - Name
- **DistinctType**
  - Extends UserDefinedType
  - Is backed by another LogicalType.
  - Family: DISTINCT
  - Characteristics:
    - Source data type
  - Future work for Flink 1.9+
- **StructuredType**
  - Extends UserDefinedType
  - Can reference an existing Java class for input and output conversion. In this regard, structured types are similar to DataType but this is required in order to also call methods on those types in the future.

- Internal data structures are up to the planner.
- Types without a referenced class need a new generic type similar to `org.apache.flink.types.Row` but with by field name access. Or we use `Row` for simplification, this is up for discussion.
- Family: STRUCTURED
- Characteristics:
    - Supertype (Java `extends`)
    - Named logical subtypes (POJO members)
    - Final/Non-final (Java `final`)
    - Instantiable/Non-instantiable (Java `abstract`)
    - Comparison properties (EQUALS/FULL/NONE)
    - Ordering properties (NONE for now)
    - Optional: reference Java class
- Methods and transform functions are out of scope. However, there is the intention of supporting this in Calcite soon (CALCITE-2721)

**Notes to Structured Types:**

- We provide tools that create structured types using Java reflection.
- Extraction will happen very conservatively:
    - *Class characteristics*:
        - No type variables or generics are allowed in the class signature.
        - No input type inference is performed.
        - Java interfaces are ignored.
        - Classes must be public and static with public constructor.
        - A constructor can be default or fully assigns all fields (same name and type!).
        - A class with no fields is a valid type.
    - *Field characteristics*:
        - Transient and static fields are ignored.
        - Every not ignored field XYZ must either:
            - be declared public
            - must have public getXYZ()/isXYZ()/XYZ() and setXYZ()/XYZ(...)
            - must have public getXYZ()/isXYZ()/XYZ() and a non-default constructor that fully assigns all fields
        - A non-default constructor that fully assigns all fields determines the order the fields otherwise we assume alphabetical ordering.
    - *Extracted type characteristics*:
        - Only the SQL types mentioned before are extracted.
        - The default type of a data type has highest precedence (e.g. byte[] is VARBINARY, not ARRAY).
        - The type `org.apache.flink.types.Row` is never treated as ANY type and an exception is thrown. This avoids confusion.

- The extraction strategy allows for describing the most important supported types: POJOs, Case Classes
- Because Tuples use generics, a user cannot use the reflection-based extraction and must define Tuples manually.

## Affected API

org.apache.flink.table.api.TableSchema
org.apache.flink.table.api.java.StreamTableEnvironment#toAppendStream
org.apache.flink.table.api.java.StreamTableEnvironment#toRetractStream
org.apache.flink.table.api.scala.StreamTableEnvironment#toAppendStream
org.apache.flink.table.api.scala.StreamTableEnvironment#toRetractStream
org.apache.flink.table.api.java.BatchTableEnvironment#toDataSet
org.apache.flink.table.api.scala.BatchTableEnvironment#toDataSet
org.apache.flink.table.api.scala.ImplicitExpressionOperations#cast
org.apache.flink.table.api.scala.ImplicitExpressionConversions#toDistinct
org.apache.flink.table.api.scala.ImplicitExpressionConversions#userDefinedAggFunctionConstructor
org.apache.flink.table.descriptors.Schema#field
org.apache.flink.table.sources.TableSource#getReturnType
org.apache.flink.table.sinks.TableSink#getOutputType
org.apache.flink.table.sinks.TableSink#getFieldTypes
org.apache.flink.table.sinks.TableSink#configure
org.apache.flink.table.sources.tsextractors.TimestampExtractor#getReturnType
org.apache.flink.table.api.Types
org.apache.flink.table.functions.ScalarFunction#getResultType
org.apache.flink.table.functions.ScalarFunction#getParameterTypes
org.apache.flink.table.functions.TableFunction#getResultType
org.apache.flink.table.functions.TableFunction#getParameterTypes
org.apache.flink.table.functions.AggregateFunction#getResultType
org.apache.flink.table.functions.AggregateFunction#getAccumulatorType
// TODO AggregateFunction has no custom parameter type option
org.apache.flink.table.expressions.Expression#resultType → thus all expression case classes
org.apache.flink.table.expressions.ExpressionParser
org.apache.flink.table.descriptors.DescriptorProperties
org.apache.flink.table.utils.TypeStringUtils
expressionDsl implicits

## Implementation Plan

1. Introduce initial new type class structure
    a. Add only classes that are required to support Flink’s current type system.
    b. Handle composite types as ANY for now.
    c. User-defined types come later.
2. Use the new type system for operand type inference and return type inference of expressions.
3. Expose the new type system through the API.
    a. Internally, we still use the old type system for both Flink and Blink planners.
4. Support user-defined types.
5. Gradually rework the Blink planner to work with the new type system.