Implementation Process for Substrait Dialect Extensions in MLIR

Implementation Process for Substrait Dialect Extensions in MLIR

Dalia Shaaban

Supervised by: Dr. Ingo Müller, Prof. Dr. Gustavo Alonso

October 2024 - March 2025

1 Introduction

Substrait

Substrait [6] is a cross-language specification that standardizes and serializes data compute operations (i.e., query plans), to enable seamless sharing of relational operators across diverse tools, programming languages, and environments. In modern data ecosystems, multiple tools and languages are often used, each with its own approach to handling data transformations. This type of fragmentation leads to redundancy, inefficiency, and increased maintenance overhead.

By defining data transformations once and allowing their efficient serialization, Substrait addresses the aforementioned issues and streamlines complex data workflows, eliminates redundancy, and reduces the need for platform-specific code. This enhances interoperability, making data processing more efficient, consistent and manageable across a variety of systems.

Despite best efforts to unify and standardize query plans, Substrait faces certain challenges, with a key limitation lying in how it delegates the handling of serialized data processing plans to individual target systems. Each system must independently implement the functionality to deserialize, parse, optimize, and execute intermediate representations (IRs). While Substrait can abstractly represent the logical plan - defining *what* needs to be done — each system must independently implement a concrete physical plan that specifies *how* these query plans will be executed. This work is thus duplicated across environments without a unified IR handling framework or approach.

Substrait-MLIR

The Substrait-MLIR project [4] is in its infancy, and aims to address the challenge of decentralized handling of Substrait Intermediate Representations (IRs). By leveraging

the Multi-Level Intermediate Representation (MLIR) compiler framework [1], the project aims to centralize the processing of Substrait IR's. MLIR is a modular compiler development framework, designed for developing (domain-specific) compilers, and is known for its extensibility and support for user-defined operations and types (dialects) tailored to specific application domains. MLIR's dialect structure provides a flexible framework to model domain-specific use cases within compiler infrastructure, and through its modular design, it allows different layers of abstraction to be created and reused.

Through the integration of Substrait with MLIR, this project establishes a common infrastructure to handle serialization, deserialization, error checking, and optimization of Substrait plans. By doing so, it eliminates the need for each system to independently implement these functionalities, reducing redundancy and fostering a consistent approach to handling Substrait IRs.

The initial focus of the MLIR-Substrait project is to expand coverage and ensure that the framework can support a wide range of Substrait-defined transformations. As the project evolves, it aims to enable Substrait-defined transformations to be translated into optimized execution strategies across various systems.

By centralizing the processing of Substrait IRs within the MLIR framework, this project not only reduces duplicated effort across environments but also creates a foundation for advancing the interoperability and efficiency of query plans.

2 My Deliverables

Given the early stage of this project, my contributions played a pivotal role in the development of the project and helped shape the core architecture of Substrait-MLIR interoperability. My main focus was on implementing parts of the Substrait dialect within MLIR.

- **Set Relation Operations:** Union, distinct, intersection, and difference operations.
- **Fetch Operation:** Support for retrieving limited subsets of data.
- **Join Operation:** Support for combining datasets based on specified conditions.
- **Simple Data Types:** Implementation of 14 data types:
  - Signed integers (`si8`, `si16`, `si64`)
  - Floating-point numbers (`f32`, `f64`)
  - Strings (`string`)
  - Temporal types (`time`, `date`, `timestamp`, `timestamp_tz`)
  - Binary (`binary`)
  - Intervals (`interval_year`, `interval_day`)
  - Universally Unique Identifier (`uuid`).
- **Compound Data Types:** Implementation of 2 data types:
  - Fixed-length unicode string (`FixedChar<L>`)
  - Variable-length Unicode String (`VarChar<L>`)

I contributed to the Substrait-MLIR repository by upstreaming the entire implementation of the aforementioned Substrait dialect components. To accomplish this, I authored a total of 40 commits and developed 6,126 lines of code. This includes 5,612 lines added and 514 lines removed during the development process.

These extensions aimed to expand Substrait's interoperability within MLIR, laying the groundwork for efficient processing of Substrait plans and enabling their representation and optimization within the MLIR framework.

3 Implementation Process of an Operation

To illustrate the typical process of implementing a Substrait operation, I will use the `FetchOp` as an example.

3.1 Reviewing the Substrait Specification

I began by thoroughly reviewing the Substrait specification for the `FetchOp` to understand its semantics and map its constructs to equivalent representations in MLIR.

The `FetchOp` retrieves a specific window of records by skipping a defined number of entries and selecting the next set of results. It aligns with the SQL `OFFSET` and `FETCH` clauses, where `OFFSET` skips the first y records and `FETCH` retrieves the next x records.

Table 1 , adapted from the Substrait specification, outlines the key properties of `FetchOp`: a relational `input`, an `offset` expression, and a `count` expression.

Property	**Description**	**Required**
Input	A relational input, typically with a desired orderedness property.	Required
Offset Expression	An expression which evaluates to a non-negative integer or null (recommended type is `i64`). Declares the offset for retrieval of records. An expression evaluating to null is treated as 0.	Optional, defaults to a 0 literal.
Count Expression	An expression which evaluates to a non-negative integer or null (recommended type is `i64`). Declares the number of records that should be returned. An expression evaluating to null indicates that all records should be returned.	Optional, defaults to a null literal.

Table 1: Properties of the Fetch Operation, [6]

3.2 Defining the Operation in TableGen

MLIR enables defining operations in a declarative manner using TableGen — which is both a domain-specific language and its tooling, designed to document domain-specific knowledge. In the context of MLIR, TableGen records facts about an operation, which are later expanded into an equivalent `mlir::Op` C++ template specialization at compiler build time[2].

The first concrete step I took was in the `SubstraitOps.td` file, where I defined the structure and semantics of the `FetchOp`, including its attributes, operands, and results to match the substrait specification (see Listing 1).

The operation `Substrait_FetchOp` extends the `RelOp` class, as a fetch operation is inherently relational. The `summary` and `description` fields provide basic documentation. The `arguments` field, specifies three inputs - `input`, `offset`, and `count` - which correspond to the fetch operation's properties as defined by the substrait specification.

- The `input` is of type `Substrait_Relation`, representing the Substrait dialect's abstraction of a relation.
- For the `offset`, I chose to model it as a constant attribute (in MLIR `attr`) of MLIR type `I64Attr` which maps to the `i64` type, following the specification's recommendation. To enforce that the offset is non-negative, the attribute is wrapped in a `ConfinedAttr` with the `IntNonNegative` constraint. Additionally, I used `DefaultValuedAttr` to assign a default value of `0` if no offset is specified—reflecting the specification's optional nature of this field.
- Similarly, the `count` argument is defined as an `I64Attr` wrapped in a `ConfinedAttr`, which ensures the attribute has a minimum value of `-1`, in alignment with the specification.

The `results` field specifies the return type of the `FetchOp`, which, by design, matches the type of the `input` — a `Substrait_Relation`. This reflects that applying the fetch operation to a relation produces another relation.

Finally, the `assemblyFormat` field defines how the operation is printed and parsed in MLIR's custom assembly form. By default, MLIR generates generic assembly syntax, but to improve readability and align more closely with the semantics of a fetch operation, I decided to implement a custom parser and printer. I thought that making the operation resemble SQL-like syntax would make it more intuitive for users.

The custom assembly format is designed as follows:

```
fetch <count> offset <offset> from <input> : <input_type>
```

For example:

```
fetch 5 offset 3 from %0 : tuple<si32>
```

In this format:

- `fetch <count>` specifies how many records to retrieve.
- `offset <offset>` (optional) specifies how many records to skip - defaults to zero if omitted.
- `from <input>` indicates the source relation operand.
- `: <input_type>` specifies the type of the input relation for type verification.

Listing 1: Definition of `Substrait_FetchOp` in TableGen

```
1  def Substrait_FetchOp : Substrait_RelOp<"fetch", [
2      SameOperandsAndResultType
3  ]> {
4    let summary = "Fetch operation";
5    let description = [{
6      Represents a `FetchOp` message together with the `RelCommon` and input
7      `Rel` message it contains.
8    }];
9    let arguments = (ins
10     Substrait_Relation:$input,
11     DefaultValuedAttr<ConfinedAttr<I64Attr, [IntNonNegative]>, "0">: $offset,
12     ConfinedAttr<I64Attr, [IntMinValue<-1>]>: $count
13   );
14   let results = (outs Substrait_Relation:$result);
15   let assemblyFormat = [{
16     custom<CountAsAll>($count) (`offset` $offset^)? `from` $input
17         attr-dict `:` type($input)
18   }];
19 }
```

This design allows the `Substrait_FetchOp` to accurately reflect the semantics defined in the specification, ensuring both correctness and clarity in its MLIR representation.

3.3 Implementing Type Inference

The next step would typically involve implementing type inference, which is done in the `Substrait.cpp` file. This ensures correct type propagation within operations and validates type correctness.

However, for the `FetchOp`, explicit type inference was unnecessary. By declaring the operation with the `SameOperandsAndResultType` trait (see line 2 of Listing 1), MLIR automatically infers the result type based on the input. In contrast, for operations without such a trait, it is necessary to implement a dedicated `<NameOfOperation>Op::inferReturnTypes` function in `Substrait.cpp` to handle type inference explicitly.

3.4 Handling Translation (Protobuf - MLIR)

Substrait uses Protocol Buffers (Protobuf) as its serialization format to define and exchange query plans. All relational operators (e.g., `FetchRel`, `JoinRel`, `SetRel`) are represented as Protobuf messages defined in Substrait's `.proto` files.

The Substrait specification defines the schema of `FetchRel` in Protobuf, detailing the structure and format of the message (see Listing 2).

Listing 2: `FetchRel` Message Definition

```
message FetchRel {
  RelCommon common = 1;
  Rel input = 2;
  oneof offset_mode {
    int64 offset = 3;
    Expression offset_expr = 5;
  }
  oneof count_mode {
    int64 count = 4;
    Expression count_expr = 6;
  }
}
```

To enable seamless conversion between Protobuf and MLIR, I modified both the `export.cpp` and `import.cpp` files. These changes facilitate the translation of the `FetchOp` between its MLIR representation and its Protobuf equivalent. Specifically, `export.cpp` handles the conversion from the Substrait MLIR dialect to the Protobuf representation, while `import.cpp` performs the reverse.

Listing 3 presents an excerpt from my implementation of the `SubstraitExporter` class, which drives the export from MLIR to Protobuf. Similarly, Listing 4 shows a snippet of the `importFetchRel` function, which imports the Protobuf `FetchRel` message back into the Substrait MLIR dialect.

3.5 Testing

As a final step, I implemented extensive tests, including round-trip tests, to ensure the correctness and reliability of the `FetchOp` and its integration. These tests verified that the operation behaved as expected across various translations.

The round-trip tests specifically validated the following functionality:

- **Serialization:** Ensured that MLIR operations were correctly translated into their Protobuf representation.
- **Deserialization:** Verified that Protobuf messages could be accurately reconstructed back into equivalent MLIR operations.

In MLIR, `FileCheck` [3] is commonly used to validate these transformations. Listing 5 shows a snippet of one of the tests I wrote to verify the export process. Lines 23–29 contain Substrait dialect code that uses the `FetchOp` (specifically on line 26). Lines 1–7 define the round-trip test flow, specifying that the Substrait dialect is translated to Protobuf, back to Substrait, and then re-exported to Protobuf. Lines 9–21 (`CHECK`-lines) define the expected Protobuf output.

Listing 3: Exporting `FetchOp` to `FetchRel`

```
FailureOr<std::unique_ptr<Rel>> SubstraitExporter::exportOperation(FetchOp op) {
  // Build `RelCommon` message.
  auto relCommon = std::make_unique<RelCommon>();
  auto direct = std::make_unique<RelCommon::Direct>();
  relCommon->set_allocated_direct(direct.release());

  // Build `input` message.
  auto inputOp = llvm::dyn_cast_if_present<RelOpInterface>(op.getInput().
      getDefiningOp());
  if (!inputOp)
    return op->emitOpError("input was not produced by Substrait relation op");

  FailureOr<std::unique_ptr<Rel>> inputRel = exportOperation(inputOp);
  if (failed(inputRel))
    return failure();

  // Build `FetchRel` message.
  auto fetchRel = std::make_unique<FetchRel>();
  fetchRel->set_allocated_common(relCommon.release());
  fetchRel->set_allocated_input(inputRel->release());
  fetchRel->set_offset(op.getOffset());
  fetchRel->set_count(op.getCount());

  // Build `Rel` message.
  auto rel = std::make_unique<Rel>();
  rel->set_allocated_fetch(fetchRel.release());

  return rel;
}
```

Listing 4: Importing `FetchRel` to `FetchOp`

```
static mlir::FailureOr<FetchOp> importFetchRel(ImplicitLocOpBuilder builder,
                                               const Rel &message) {
  const FetchRel &fetchRel = message.fetch();

  // Import input.
  const Rel &inputRel = fetchRel.input();
  mlir::FailureOr<RelOpInterface> inputOp = importRel(builder, inputRel);

  // Build `FetchOp`.
  Value inputVal = inputOp.value()->getResult(0);
  return builder.create<FetchOp>(inputVal, fetchRel.offset(), fetchRel.count())
      ;
}
```

Listing 5: FileCheck Test for FetchOp Export

```
1  // RUN: substrait-translate -substrait-to-protobuf %s \
2  // RUN: | FileCheck %s
3
4  // RUN: substrait-translate -substrait-to-protobuf %s \
5  // RUN: | substrait-translate -protobuf-to-substrait \
6  // RUN: | substrait-translate -substrait-to-protobuf \
7  // RUN: | FileCheck %s
8
9  // CHECK-LABEL: relations {
10 // CHECK-NEXT: rel {
11 // CHECK-NEXT: fetch {
12 // CHECK-NEXT: common {
13 // CHECK-NEXT: direct {
14 // CHECK-NEXT: }
15 // CHECK-NEXT: }
16 // CHECK-NEXT: input {
17 // CHECK-NEXT: read {
18 // CHECK-NEXT: }
19 // CHECK-NEXT: }
20 // CHECK: offset: 3
21 // CHECK-NEXT: count: 5
22
23 substrait.plan version 0 : 42 : 1 {
24   relation {
25     %0 = named_table @t1 as ["a"] : tuple<si32>
26     %1 = fetch 5 offset 3 from %0 : tuple<si32>
27     yield %1 : tuple<si32>
28   }
29 }
```

This test ensures that when the Substrait dialect snippet (lines 23–29) is round-tripped, it produces the exact Protobuf representation (lines 9–21) — thus confirming the correctness of both the export and import logic.

4 Implementation Process of a Type

To illustrate the typical process of implementing a Substrait type, I will use the `interval_year` type as an example.

1. Reviewing the Substrait Specification

I began by thoroughly reviewing the Substrait specification for the `interval_year` type to understand its semantics and map its constructs to equivalent representations in MLIR.

As shown in Table 2, the `interval_year` type uses the format x`y` z`m`, where `x` represents the number of years and `z` the number of months. This type supports a range of $[-10,000..10,000]$ years or $[-120,000..120,000]$ months. The years and months should be stored as `int32` values.

Type Name	**Description**	**Protobuf representation for literals**
`interval_year`	Interval year to month. Supports $[-10,000 ... 10,000]$ years with month precision ($= [-120,000 ... 120,000]$). Stored as separate `int32` years and months, but only the total months are significant (e.g., `1y 0m`, `0y 12m`, and `1001y -12000m` are equivalent). Each component must not exceed 10,000 years, and opposite signs are disallowed (e.g., `-10000y 200000m` is **not** allowed).	`int32` years and `int32` months

Table 2: `interval_year` Type Specification [5]

2. Defining the Type in TableGen

Similarly to how operations are defined in TableGen files, types are also specified within TableGen files. The two key files involved in this process are `SubstraitTypes.td` and `SubstraitAttrs.td`.

The first step I took was editing the `SubstraitTypes.td` file, where I defined the structure and semantics of the `IntervalYearMonthType`. As shown in Listing 6, the `IntervalYearMonthType` extends the `Substrait_Type` class, and is declared along with its `summary` and `description` fields to provide context and documentation.

Listing 6: Definition of `Substrait_IntervalYearMonthType`

```
def Substrait_IntervalYearMonthType : Substrait_Type<"IntervalYearMonth", "
    interval_year_month"> {
  let summary = "Substrait␣interval␣year␣to␣month␣type";
  let description = [{
    This type represents a substrait interval year to month type.
  }];
}
```

Once the type is defined, it is added to the `Substrait_SimpleTypes` list, which aggregates all defined types. This inclusion is illustrated on line 17 of Listing 7.

Listing 7: Definition of `Substrait_SimpleTypes`

```
/// Currently supported simple types, listed in order of the Substrait
    specification.
def Substrait_SimpleTypes {
  list<Type> types = [
    SI1, // Boolean
    SI8, // I8
    SI16, // I16
    SI32, // I32
    SI64, // I64
    F32, // FP32
    F64, // FP64
    Substrait_StringType, // String
    Substrait_BinaryType, // Binary
    Substrait_TimestampType, // Timestamp
    Substrait_TimestampTzType, // TimestampTZ
    Substrait_DateType, // Date
    Substrait_TimeType, // Time
    Substrait_IntervalYearMonthType, // IntervalYear
    Substrait_IntervalDaySecondType, // IntervalDay
    Substrait_UUIDType // UUID
  ];
}
```

The second step involved editing the `SubstraitAttrs.td` file, where I defined the structure and semantics of the `IntervalYearMonth` attribute. It is important to distinguish between a type and an attribute: a type describes the set of possible values a value may hold at runtime, along with its behavior. In contrast, an attribute represents a specific, constant value known at compile time.

As shown in Listing 8 , the `Substrait_IntervalYearMonthAttr` extends the `Substrait_StaticallyTypedAttr` class. The `summary` and `description` fields provide documentation. The `parameters` field declares two arguments - `years` and `months` - to match the substrait specification.

Additionally, the `assemblyFormat` is designed to match the specification's textual format, allowing instances to be written as e.g. `3y 1m`. The `genVerifyDecl` field is set to `1`, enabling the generation of a custom verification function. This function, implemented in `Substrait.cpp`, ensures correctness by validating that the parameter values are within the specified ranges.

This verification function is implemented in the `Substrait.cpp` file. According to the Substrait specification, the `year` parameter must be within the range [−10,000..10,000] years, or equivalently, the `month` parameter must lie within [−120,000..120,000] months. As shown in Listing 9, the `verify` function enforces these constraints by checking the validity of the input values.

Listing 8: Definition of `Substrait_IntervalYearMonthAttr`

```
def Substrait_IntervalYearMonthAttr
  : Substrait_StaticallyTypedAttr<"IntervalYearMonth", "interval_year_month",
                                  "IntervalYearMonthType"> {
  let summary = "Substrait␣interval␣year␣to␣month␣type";
  let description = [{
    This type represents a substrait interval year to month attribute type.
    Note that this attribute does not attempt to canonicalize equivalent year-
        month intervals.
  }];
  let parameters = (ins "int32_t":$years, "int32_t":$months);
  let assemblyFormat = [{ `<` $years `` `y` $months `` `m` `>` }];
  let genVerifyDecl = 1;
}
```

Listing 9: Definition of `verify` function

```
LogicalResult mlir::substrait::IntervalYearMonthAttr::verify(
    llvm::function_ref<mlir::InFlightDiagnostic()> emitError,
    int32_t year,
    int32_t month) {

    if (year < -100000 || year > 100000)
        return emitError() << "year␣must␣be␣in␣a␣range␣of␣[-10,000..10,000]␣
            years";

    if (month < -120000 || month > 120000)
        return emitError()
            << "month␣must␣be␣in␣a␣range␣of␣[120,000..120,000]␣months";

    return success();
}
```

Listing 10: Definition of `Substrait_SimpleAttributes`

```
1  // Attributes of currently supported atomic types, listed in order of substrait
       specification.
2  def Substrait_SimpleAttributes {
3     list<Attr> attrs = [
4        SI1Attr, // Boolean
5        SI8Attr, // I8
6        SI16Attr, // I16
7        SI32Attr, // I32
8        SI64Attr, // I64
9        F32Attr, // FP32
10       F64Attr, // FP64
11       TypedStrAttr<Substrait_StringType>, // String
12       TypedStrAttr<Substrait_BinaryType>, // Binary
13       Substrait_TimestampAttr, // Timestamp
14       Substrait_TimestampTzAttr, // TimestampTZ
15       Substrait_DateAttr, // Date
16       Substrait_TimeAttr, // Time
17       Substrait_IntervalYearMonthAttr, // IntervalYear
18       Substrait_IntervalDaySecondAttr, // IntervalDay
19       Substrait_UUIDAttr, // UUID
20    ];
21 }
```

Finally, similarly as in the type case, the `Substrait_IntervalYearMonthAttr` is added to the `Substrait_SimpleAttributes` list, which aggregates all defined attributes. This inclusion is illustrated on line 17 of Listing 10.

3. Enabling Translation (MLIR - Protobuf)

To enable seamless conversion between Protobuf and MLIR, I modified both the `export.cpp` and `import.cpp` files, following a similar approach to the implementation of operations. These changes support the translation of the `IntervalYearMonthType` between its MLIR representation and corresponding Protobuf message. Specifically, `export.cpp` handles the conversion from the Substrait MLIR dialect to Protobuf, while `import.cpp` performs the reverse translation.

Listing 11 shows the implementation of the `exportOperation` function, which drives the export process from the Substrait dialect to Protobuf. Using the literal operations' attribute and type, the code uses conditional checks to determine whether it is a `IntervalYearMonthType`. If it is, an "empty" Protobuf message for this type is declared in lines 14 and 15. The function then extracts the relevant information - specifically, the year and month values (lines 17 and 18) - and sets these fields in the Protobuf representation (lines 19 to 22).

Listing 12 shows the implementation of the `importLiteral` function, which handles the import process for types from the Protobuf representation back into the Substrait MLIR dialect. The implementation and logic of this function follows a similar structure to the examples discussed throughout this document.

4. Testing

Finally, I developed comprehensive tests to verify the correctness and robustness of the `IntervalYearMonthType`. As with the operation tests, `FileCheck` was used to validate the results. Listing 13 presents a snippet of one such test designed to verify the import process of the `IntervalYearMonthType`.

In this test, a Protobuf message containing the type (see line 55) is translated into the Substrait dialect, then converted back to Protobuf, and finally re-imported into Substrait. The resulting output is compared against the expected values specified alongside the `CHECK` statements.

Listing 11: Exporting Substrait_IntervalYearMonthType to IntervalYearToMonth

```
FailureOr<std::unique_ptr<Expression>>
SubstraitExporter::exportOperation(LiteralOp op) {
  // Build `Literal` message depending on type.
  Attribute value = op.getValue();
  mlir::Type literalType = getAttrType(value);
  auto literal = std::make_unique<Expression::Literal>();

  // `IntegerType`s.
  if (auto intType = dyn_cast<IntegerType>(literalType)) {
    ....
  }
  // `IntervalType`s.
  else if (auto intervalType = dyn_cast<IntervalYearMonthType>(literalType)) {
    auto intervalYearToMonth = std::make_unique<
        ::substrait::proto::Expression_Literal_IntervalYearToMonth>();
    auto intervalYearMonth = mlir::cast<IntervalYearMonthAttr>(value);
    int32_t intervalYear = intervalYearMonth.getYears();
    int32_t intervalMonth = intervalYearMonth.getMonths();
    intervalYearToMonth->set_years(intervalYear);
    intervalYearToMonth->set_months(intervalMonth);
    literal->set_allocated_interval_year_to_month(intervalYearToMonth.release())
        ;
  } else if (...)
  ....
  } else
    op->emitOpError("has␣unsupported␣value");

  // Build `Expression` message.
  auto expression = std::make_unique<Expression>();
  expression->set_allocated_literal(literal.release());

  return expression;
}
```

Listing 12: Importing Substrait kIntervalYearToMonth to Substrait_IntervalYearMonthType

```
static mlir::FailureOr<LiteralOp>
importLiteral(ImplicitLocOpBuilder builder,
              const Expression::Literal &message) {
  MLIRContext *context = builder.getContext();
  Location loc = builder.getLoc();

  Expression::Literal::LiteralTypeCase literalType =
      message.literal_type_case();

  switch (literalType) {
      case Expression::Literal::LiteralTypeCase::kI16: {
        ....
      }
      case Expression::Literal::LiteralTypeCase::kIntervalYearToMonth: {
        auto attr = IntervalYearMonthAttr::get(
            context, message.interval_year_to_month().years(),
            message.interval_year_to_month().months());
        return builder.create<LiteralOp>(attr);
      }
  }
}
```

Listing 13: FileCheck Test for IntervalYearMonth Import (shortened for readability

```
# RUN: substrait-translate -protobuf-to-substrait %s \
# RUN: --split-input-file="#␣""-----" \
# RUN: | FileCheck %s

# RUN: substrait-translate -protobuf-to-substrait %s \
# RUN: --split-input-file="#␣""-----" --output-split-marker="//␣-----" \
# RUN: | substrait-translate -substrait-to-protobuf \
# RUN: --split-input-file --output-split-marker="#␣""-----" \
# RUN: | substrait-translate -protobuf-to-substrait \
# RUN: --split-input-file="#␣""-----" --output-split-marker="//␣-----" \
# RUN: | FileCheck %s

# CHECK: substrait.plan version 0 : 42 : 1 {
# CHECK-NEXT: relation
# CHECK: %[[V0:.*]] = named_table
# CHECK-NEXT: %[[V1:.*]] = project %[[V0]] : tuple<si1> -> tuple<si1, !
    substrait.interval_year_month, !substrait.interval_day_second> {
# CHECK-NEXT: ^[[BB0:.*]](%[[ARG0:.*]]: tuple<si1>):
# CHECK-NEXT: %[[V2:.*]] = literal #substrait.interval_year_month<2024y 1m>
# CHECK-NEXT: %[[V3:.*]] = literal #substrait.interval_day_second<9d 8000s>
# CHECK-NEXT: yield %[[V2]], %[[V3]] : !substrait.interval_year_month, !
    substrait.interval_day_second
# CHECK-NEXT: }
# CHECK-NEXT: yield %[[V1]] : tuple<si1, !substrait.interval_year_month, !
    substrait.interval_day_second>

relations {
  rel {
    project {
      common {
        direct {
        }
      }
      input {
        read {
        ...
        }
      }
      expressions {
        literal {
          interval_year_to_month {
            years: 2024
            months: 1
          }
        }
      }
      expressions {
        ...
        }
      }
    }
  }
}
version {
  minor_number: 42
  patch_number: 1
}
```

5 Other contributions and tasks

In addition to the major deliverables mentioned above, I performed various other tasks throughout the project. These included:

- Adding custom parsers and printers where necessary to handle specific use cases.
- Conducting general cleanup work to improve code readability and maintainability.
- Addressing miscellaneous issues to ensure the functionality of the Substrait-MLIR framework.

6 Conclusion

My work focused on extending the Substrait dialect within MLIR. By implementing core operations and data types, I adressed key gaps in the Substrait-MLIR integration.

7 References

[1] Chris Lattner et al. *MLIR: Scaling Compiler Infrastructure for Domain Specific Computation*. Accessed: March 19, 2025. 2021. DOI: 10.1109/CGO51591.2021.9370308.

[2] MLIR Project. *Defining Dialects and Operations*. Accessed: March 19, 2025. 2025. URL: https://mlir.llvm.org/docs/DefiningDialects/Operations/.

[3] *MLIR Testing Guide*. Accessed: March 19, 2025. URL: https://mlir.llvm.org/getting_started/TestingGuide/.

[4] Substrait Project. *Substrait MLIR Contrib*. URL: https://github.com/substrait-io/substrait-mlir-contrib.

[5] Substrait Project. *Substrait Type Classes*. Accessed: March 19, 2025. 2025. URL: https://substrait.io/types/type_classes/.

[6] Substrait Project. *Substrait: Cross-language specification for data compute operations*. Accessed: March 19, 2025. URL: http://substrait.io.