Spark VStream Format

Version 1.0

|  |  |
| --- | --- |
| Revision History |  |
| Rev 1.0 | 12-20-2019, raymcc, Describing Version 1.0 of VStream format |
|  |  |

## Abstract

This paper defines VStream, a useful binary encoding used by the family of C++ based “Vegas” services and tools.

VStream is a **binary row-oriented format** designed to interoperate efficientwith the JVM and Scala and to be easy to implement in C++ runtime or FPGA processor. The primary use cases are to represent pre-parsed row oriented data such as JSON or CSV data and to act as a common network streaming format. It is not intended to compete with Arrow or Parquet, which are efficient column-oriented encodings.

The general goal is to be able to construcct a Spark InternalRow object using data read from a VStream with little or no additional JVM-based computation, parsing or allocation. Services and processes such as the Vegas Cache service and FPGA co-processors can emit this format so that the row data can be directly piped into Spark with little or no transformation.

## Introduction

The VStream format is a “block” format, in which each “block” contains a number of logical “rows”. In turn, the “row” consists of packed columnar data, which is usually the in-memory form in the C++ runtime, adjusted for big-endian orientation of the target is the JVM/Scala/Spark runtime. The VStream format is thus sequential and self-relative and can be directly deliver via any I/O mechanism such as a file or socket or shared memory by simply performing a bitwise copy.

The format was designed to be stored in a file or read from a socket, and because it is sequential, is streaming-compatible.

For example, the FPGA processor will process JSON or CSV files and produce this binary format which is directly supported by the Spark Cache Services and Vegas DSV2 plug-in. After the initial download from Azure Data Lake and processing, the Spark DSV2 can re-read the VStream format quickly to avoid secondary parsing of the data, representing a significant throughput increase for Spark even beyond merely caching the file locally, since the parsing step never needs to be repeated.

Note that the Spark cache, being able to work with this format, can cache pre-parsed content and deliver it to the DSV2 implementation to avoid re-parsing, or pre-filtered content, as the predicate push down filtering mechanism emits this format as well.

## VStream Block Layout

The VStream format is “packed” in that there is no padding. Each token is written and the subsequent token begins at the very next byte in the stream. The delimiter tokens are 32 bit constants and always paired:

*VStream\_Begin\_Block*

*VStream\_Begin\_Block\_Header*

*…metadata… such as the little/big-endian orientation and schema reference*

*VStream\_End\_Block\_Header*

*VStream\_Begin\_Row*

*Col0 Col1 … Col N packed binary data…*

*VStream\_End\_Row*

*VStream\_Begin\_Row*

*Col0 Col1 … Col N packed binary data…*

*VStream\_End\_Row*

*VStream\_End\_Block*

NOTES:

1. **The begin/end tokens are always 32 bits and endian-agnostic.** They are thus the same constant value in little endian as big endian (palindromes, in essence). This makes it easier to debug.
2. These tokens are used as a delimiter and verification mechanism to prove integrity.
3. Tokens are endian-agnostic and are thus the same value in both the Java runtime (Scala) and the C++ runtime, even though the JVM has big-endian orientation. For example:

VTOKEN\_BEGIN\_ROW 0x71A0A071

VTOKEN\_END\_ROW 0x72A0A072

## VStream Files

A persisted VStream file may consist of a single block, or successive blocks placed end to end. It is legal to simply concatenate a VStream block to an existing file consisting of one or more blocks. The common extension is .vstream.

## VStream Block

**A VStream Row Block** is laid out as follows:

|  |  |  |
| --- | --- | --- |
| **Token** | **Value** | **Notes** |
| **VSTREAM\_BEGIN\_BLOCK** | 0x73A0A073 | Indicates this is a VStream block |
| **VSTREAM\_ENDIAN\_FLAG** | 0x10101010 Little Endian 0x11111111 Big Endian | The encoding used from this point on |
| **VSTREAM\_SCHEMA\_REF** | UTF8 string | Typically a JSON value indicating the schema implied by the rows in this block; may be a zero length string. Encoding is as for UTF8 string in Section 8 of this specification. |
| ***VStream Row Array*** | A set of VStream rows |  |
|  |  |  |
| **VSTREAM\_END\_BLOCK** | 0x74A0A074 | Indicates the end of the block |

Note that the tokens are endian-agnostic. However, once big-endian or little-endian is specified, all content in the block from that point on must match the specified encoding, where needed. This applies to all multi-byte types such as Int16, Int32, Int64, GUID/UUID and the integer length prefix on strings.

## VStream Row

**A VStream Row** is laid out as follows:

|  |  |  |
| --- | --- | --- |
| **Token** | **Value** | **Notes** |
| **VSTREAM\_BEGIN\_ROW** | 0x71A0A071 | Indicates a row follows |
| **…column values…** |  |  |
| **VSTREAM\_END\_ROW** | 0x72A0A072 | Indicates the end of a row |

The encoding for a row consists of the VSTREAM\_BEGIN\_ROW token, the binary encoding of the column data, followed by the VSTREAM\_END\_ROW token.

For example, to encode row consisting of three 32 bit integers (1, 2, 3), the binary layout would be as follows (little endian):

|  |  |  |
| --- | --- | --- |
| Offset | Value | Meaning |
| 0 | VSTREAM\_BEGIN\_ROW (4 bytes) | Row header |
| 4 | 1 (4 bytes) 0x1 0x0 0x0 0x0 | First integer |
| 8 | 2 (4 bytes) 0x2 0x0 0x0 0x0 | Second integer |
| 12 | 3 (4 bytes) 0x3 0x0 0x0 0x0 | Third integer |
| 16 | VSTREAM\_END\_ROW (4 bytes) | End row marker |
| 20 | … |  |

Where the integer values are adjusted for little/big-endian.

Another example with a 32 bit integer, a string “ABC”, an a BOOL set to *true:*

|  |  |  |
| --- | --- | --- |
| Offset | Value | Meaning |
| 0 | VSTREAM\_BEGIN\_ROW | Row header |
| 4 | 1 | First integer |
| 8 | 3 | Length of string (8 bytes) |
| 16 | ‘A’ | First char of string |
| 17 | ‘B’ | Second char |
| 18 | ‘C’ | Third char |
| 19 | 1 | Bool ‘true’ |
| 20 | VSTREAM\_END\_ROW | End marker |
| 24 | …. |  |

All the integer values including the string length must be little/big-endian adjusted.

## VStream Column Values

VStream column values are literal constants written back to back. These must be adjusted to match the endian orientation that is being encoded. The implied schema of the various columns is described separately in a VStream Schema block.

The TypeCode constants are themselves only used in a few cases and in schema definition blocks, but are presented for reference purposes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Type Code** | **Format** | **Notes** |
| **Int32** | TypeCode\_Int32 | 32 bit signed integer | Subject to endian rules |
| **Int64** | TypeCode\_Int64 | 64 bit signed integer | Subject to endian rules |
| **UTF8 String** | TypeCode\_UTF8 | 64 bit length prefix, followed by that many characters | The prefix is subject to the endian encoding, but the characters are individual bytes and are serialized the same way in either case |
| **UTF16 String** | TypeCode\_UTF16 | 64 bit length prefix incharactercount, followed by the 16 bit UTF characters. | Note that the length prefix is subject to endian rules and since the characters are 16 bit, they too must be endian adjusted. The length prefix is the character count, so the byte count is twice that. |
| **UTF32 String** | TypeCode\_UTF32 | 64 bit length prefix, followed by that many UTF32 characters | Note that the length prefix is subject to endian rules and since the characters are 32 bit, they too must be endian adjusted. The length prefix is the character count, so the byte count is 4x the length. |
|  |  |  |  |
| **NULL** | TypeCode\_Null | 64 bit value of all zeros | Used to represent database NULL, JSON null. |
| **BOOL** | TypeCode\_Bool | 1 byte, with the value 1 for TRUE or 0 for false. |  |
|  |  |  |  |
| **TIMESTAMP (uint64\_t)** | TypeCode\_Timestamp | 64 bit value of Unix epoch nanoseconds | See DateTime encoding in Appendix |
| **Byte** | TypeCode\_Byte | Literal byte value 0..255 |  |
| **Char8** | TypeCode\_Char8 | Literal 8 bit character value |  |
| **Char16** | TypeCode\_Char16 | Literal UTF16 character | Subject to endian rules |
| **Char32** | TypeCode\_Char32 | Literal UTF32 character | Subject to endian rules |
| **Int16** | TypeCode\_Int16 | 16 bit signed integer | Subject to endian rules |
| **Uint16** | TypeCode\_UInt16 | 16 bit unsigned integer | Subject to endian rules |
| **UInt32** | TypeCode\_UInt32 | 32 bit unsigned integer | Subject to endian rules |
| **UInt64** | TypeCode\_UInt64 | 64 bit unsigned integer | Subject to endian rules |
| **UUID** | TypeCode\_UUID | 16 octet RFC 4122 UUID | Subject to endian rules; |
| **Double** | TypeCode\_Double | IEEE 8 byte double precision encoding | Subject to endian rules |
| **Float** | TypeCode\_Float | IEEEE 4 byte double precision encoding | Subject to endian rules |
| **ISO8601** | TypeCode\_ISO8601 | Date Time encoded as ISO8601 String | Uses UTF8 string rules |
| **JSON** | TypeCode\_JSON | JSON fragment as UTF8 string | Uses UTF8 string rules |
| **Vector** | TypeCode\_Vector | See special encodings | TBD |
| **Map** | TypeCode\_Map | See special encodings | TBD |
| **Tuple** | TypeCode\_Tuple | See special encodings | TBD |
| **Timestamp96** | TypeCode\_Timestamp96 | See special encodings | TBD: Parquet 96 bit timestamp, i.e., Julian Day + 64 bit nanosecond offset. |

## VStream Schema

VStream schema is described in JSON. There are two types, a schema definition, and a schema reference. In most cases, VStream Block Headers contain the smaller form of JSON which contains a reference to the larger format with the full definition.

A schema definition is a JSON document with two required fields, plus any other addition ad-hoc fields that the user may need to indicate origin, author, version, etc.:

{  
 “VStream.Schema.Id” : “Any string, but usually a URI or a GUID”,  
 “VStream.Column.Definitions” :   
 {  
 “Column1” : 3, // The value is the typecode constant.  
 “Column2” : 2

},  
 …

}

The VStream.Schema.Id value is any string, but should be something unambiguous such as a URI or a GUID.   
The column definitions are value pairs containing the column name and the integeral value of a TypeCode\_ constant. The column order is the same ordering used in the VStream Row blocks.

A schema reference only contains the ID portion:

{  
 “VStream.SchemaRef” : “The Id of schema definition”,  
 …  
}

## Appendix 1: FPGA Encoding Requirements

The FPGA processor is not required to support the full set of value types, but must support the following subset.

The FPGA interface runs in two modes, “Specified Schema” and “Inferred Schema”. When Specified Schema is used, the calling side (Cache Service) will define the schema for each column type. The required types that must be supported are:

1. Int32
2. Int64and als
3. UTF8String
4. Timestamp
5. Float
6. Double
7. Bool
8. JSON

When inferred schema is used, the supported types are fewer, since there is no way to know if certain numeric types are supposed to be the long or short forms:

1. Int64
2. UTF8String
3. Timestamp
4. Double
5. Bool

In other words, when the FPGA processor encounters 3.14, it should assume the value is a Double. If the value 7 is encountered, it will be assumed to be a signed 64 bit integer.

The second case occurs for JSON lines where a JSON value is embedded JSON fragment. In those cases, the column should be a JSON column type with the fragment contained within.

For example, this JSON Line

{ “C1” : 123, “C2” : 456, “C3”: { “A” : 12, “B” : 13 }, “C4” : 789 }

...encodes to 4 columns:

1. C1 = 123 (Int64)
2. C2 = 456 (Int64)
3. C3 = “{ “A”: 12, “B” : 13 }” (Json column)
4. C4 = 789 (Int64)

## Appendix 2 : Timestamp Encoding

Timestamps are represented as a 64 bit unsigned value which contains nanoseconds since the Unix Epoch, Jan 1. 1970. As such this will overflow in 584 years, so there is plenty of room as of this writing. [😊].

*NOTE: This value is compatible with the C++ standard high\_resolution\_clock on both Windows and Linux platforms, and the system\_clock on Linux. The system\_clock on Windows is in 100 nanosecond granularity, so the value must be adjusted or elseuse the high\_resolution clock used when interoperating with Windows platforms.*

The reason for this is to be fully compatible with all current systems (which all use the Unix epoch), such as the Linux VM system clocks, and not lose any of the fractional value part during coding/decoding. In systems which don’t have this granularity it is trivial to convert down to whatever granularity is required. For example, Java.Sql.DateTime has a constructor based on milliseconds, this value can be divided by 1,000,000 and used for that purpose.

When parsing text-based dates, such as ISO 8601 text timestamps, for example, the fractional second field may be present or missing:

2019-12-29 14:00:21 // No fractional timestamp

2019-12-29 14:00.21.051 // 51 milliseconds

2019-12-29 14:00.21.000512 // 512 microseconds

In these cases, the parser can extract the fields and uses mktime to produce the standard time\_t, which brings the date to a single value, the number of seconds since Jan 1, 1970.

This would then be multipled to its nanosecond equivalent and the fractional part added in, if present, after adjusting it to a nanosecond value:

Uint64\_t Timestamp = time\_t \* 1000000000000 // Convert from seconds to nanoseconds

Timestamp += fractional\_part\_in\_nanos

The timezone is not specified, and is inferred by code using the timestamp. In the event that the timezone is specified (as allowed by ISO 8601), the parsing code should adjust the timestamp to UTC (GMT) and normalize it.

2019-12-29T14:00:21Z // Already GMT, just ignore the Z

However, this date should be converted

2019-12-29T14:00:00-08:00 // Two o’clock in the after noon in Pacific Daylight, but 22:00 GMT (10pm).

2019-12-29T22:00:00Z // The date should be normalized to this

The reason for this is that when queries in Spark begin comparing timestamps, they don’t know about the timezone, so there must be a normalized value.