```
["PARTIAL-DATA-REF", ["https://example.db.org/value2"]]
[[11, 110], ["PARTIAL-DATA-REF", ["https://example.db.org/value3"]], [550, 333]]
["PARTIAL-DATA-NEXT", ["https://example.db.org/value4"]]
```

Below follows an example of the sparse layout for multidimensional lists with three aggregated dimensions. The underlying property value can be taken to be sparse data in lists in four dimensions of 10000 x 10000 x 10000 x N, where the innermost list is a non-sparse list of arbitrary length of numbers. The only non-null items in the outer three dimensions are, say, [3,5,19], [30,15,9], and [42,54,17]. The response below communicates the first item explicitly; the second one by deferring the innermost list using a reference-marker; and the third item is not included in this response, but deferred to another page via a next-marker.

```
{"optimade-partial-data": {"format": "1.2.0"}, "layout": "sparse"}
[3,5,19,  [10,20,21,30]]
[30,15,9, ["PARTIAL-DATA-REF", ["https://example.db.org/value1"]]]
["PARTIAL-DATA-NEXT", ["https://example.db.org/"]]
```

An example of the sparse layout for multidimensional lists with three aggregated dimensions and integer values:

```
{"optimade-partial-data": {"format": "1.2.0"}, "layout": "sparse"}
[3,5,19,  10]
[30,15,9, 31]
["PARTIAL-DATA-NEXT", ["https://example.db.org/"]]
```

An example of the sparse layout for multidimensional lists with three aggregated dimensions and values that are multidimensional lists of integers of arbitrary lengths:

```
{"optimade-partial-data": {"format": "1.2.0"}, "layout": "sparse"}
[3,5,19, [ [10,20,21], [30,40,50] ] ]
[3,7,19, ["PARTIAL-DATA-REF", ["https://example.db.org/value2"]]]
[4,5,19, [ [11, 110], ["PARTIAL-DATA-REF", ["https://example.db.org/value3"]], [550, 333]]]
["PARTIAL-DATA-END", [""]]
```

9.5 OPTIMADE Regular Expression Format

This section defines a Unicode string representation of regular expressions (regexes) to be referenced from other parts of the specification. The format will be referred to as an "OPTIMADE regex".

Regexes are commonly embedded in a context where they need to be enclosed by delimiters (e.g., double quotes or slash characters). If this is the case, some outer-level escape rules likely apply to allow the end delimiter to appear within the regex. Such delimiters and escape rules are *not* included in the definition of the OPTIMADE regex format itself and need to be clarified when this format is referenced. The format defined in this section applies after such outer escape rules have been applied (e.g., when all occurrences of `\/` have been translated

into `/` for a format where an unescaped slash character is the end delimiter). Likewise, if an OPTIMADE regex is embedded in a serialized data format (e.g., JSON), this section documents the format of the Unicode string resulting from the deserialization of that format.

The format is a subset of the format described in ECMA-262, section 21.2.1. The format is closely inspired by the subset recommended in the JSON Schema standard, see JSON Schema: A Media Type for Describing JSON Documents 2020-12, section 6.4. However, OPTIMADE has decided to restrict the subset further to better align it with the features available in common database backends and to clarify the limitations of character classes and character escapes. The intent is that the specified format is also a subset of the PCRE2 regex format to make the format directly useful (without translation) in a wide range of regex implementations.

Hence, an OPTIMADE regex is a regular expression that adheres to ECMA-262, section 21.2.1 with the additional restrictions described in the following. The regex is interpreted according to the processing rules that apply for an expression where only the Unicode variable is set to true of all variables set by the RegExp internal slot described by ECMA-262, section 21.2.2.1. Furthermore, it can only use the following tokens and features (this list is partially quoted from the JSON Schema standard):

- Individual Unicode characters matching themselves, as defined by the JSON specification.
- The `.` character to match any one Unicode character except the line break characters LINE FEED (LF) (U+000A), CARRAGE RETURN (U+000D), LINE SEPARATOR (U+2028), PARAGRAPH SEPARATOR (U+2029) (see ECMA-262 section 2.2.2.7).
- A literal escape of one of the syntax characters, i.e., the escape character (`\`) followed by one of the following characters `^ $ \ . * + ? ( ) [ ] { } |` to represent that literal character. No other characters can be escaped. (This rule prevents other escapes that are interpreted differently depending on regex flavor.)
- Simple character classes (e.g., `[abc]`) and range character classes (e.g., `[a-z]`) with the following constraints:
  - The character `-` designates ranges unless it is the first or last character of the class.
  - The characters `\ [ ]` can only appear escaped with a preceeding backslash, e.g. `\\` designates that the class includes a literal `\` character. The other syntax characters may appear either escaped or unescaped to designate that the class includes them. (This rule prevents other escapes inside classes that are not the same across regex flavors and expressions that, in some flavors, are interpreted as nested classes.)
  - Except for as specified above, all characters represent themselves literally (including syntax characters).
  - Each literal character can appear in the class at most once. (This

rule prevents expressions interpreted as pre-defined classes in some regex flavors, e.g., `[:alpha:]`).

- Complemented character classes (e.g., `[^abc]`, `[^a-z]`).
- Simple quantifiers: `+` (one or more), `*` (zero or more), `?` (zero or one) that appear directly after a character, group, or character class. (This rule prevents expressions with special meaning in some regex flavors, e.g., `+?` and `(? )`.)
- The beginning-of-input (`^`) and end-of-input (`$`) anchors.
- Simple grouping (`(...)`) and alternation (`|`).

Note that compared to the JSON Schema standard, lazy quantifiers (`+?`, `*?`, `??`) are *not* included, nor are range quantifiers (`{x}`, `{x,y}`, `{x,}`). Furthermore, there is no support for escapes designating shorthand character classes as `\` and a letter or number, nor is there any way to represent a Unicode character by specifying a code point as a number, only via the Unicode character itself. (However, the regex can be embedded in a context that defines such escapes, e.g., in serialized JSON a string containing the character `\u` followed by four hexadecimal digits is deserialized into the corresponding Unicode character.)

An OPTIMADE regex matches the string at any position unless it contains a leading beginning-of-input (`^`) or trailing end-of-input (`$`) anchor listed above, i.e., the anchors are not implicitly assumed. For example, the OPTIMADE regex "es" matches "expression".

Regexes that utilize tokes and features beyond the designated subset are allowed to have an undefined behavior, i.e., they MAY match or not match *any* string or MAY produce an error. Implementations that do not produce errors in this situation are RECOMMENDED to generate warnings if possible.

> Compatibility notes:
>
> The definition tolerates (with undefined behavior) regexes that use tokens and features beyond the defined subset. Hence, a regex can be directly handed over to a backend implementation compatible with the subset without needing validation or translation. Additional consideration of how the `.` character operates in relation to line breaks may be required for multiline text. If the regex is applied to strings containing only the LINE FEED (U+000A) character and none of the other Unicode line break characters, most regex backend implementations are compatible with the defined behavior. If the regex is applied to string data containing arbitrary combinations of line break characters and the right behavior cannot be achieved via environmental settings and regex options, implementations can consider a translation step where other line break characters are translated into LINE FEED in the text operated on.
>
> Compatibility with different regex implementations may change depending on the environment, language versions, and options and has to be verified by implementations. However, as a general guide, we

have used third-party sources, e.g., the Regular Expression Engine Comparison Chart to collect the following information for compatibility when operating on text using LINE FEED as the line break character:

- ECMAScript (also known as javascript) and version 1 and 2 of PCRE are meant to be compatible by design.
- The following regex formats appear generally compatible when operating in Unicode mode: Perl, Python, Ruby, Rust, Java, .NET, MySQL 8, MongoDB, Oracle, IBM Db2, Elasticsearch, DuckDB (which uses the re2 library).
- SQLite supports regexes via libraries and thus can use a compatible format (e.g., PCRE2).
- XML Schema appears to use a compatible regex format, except it is implicitly anchored: i.e., the beginning-of-input `^` and end-of-input `$` anchors must be removed, and missing anchors replaced by `.*`.
- POSIX Extended regexes (and their extended GNU implementations) are incompatible because `\` is not a special character in character classes. POSIX Basic regexes also have further differences, e.g., the meaning of some escaped syntax characters is reversed.