OData Version 4.01. Part 2: URL Conventions

Working Draft ~~01~~02

08 June~~08 December~~ 201~~6~~7

Technical Committee:

OASIS Open Data Protocol (OData) TC

Chairs:

Ralf Handl (ralf.handl@sap.com), SAP SE
Michael Pizzo (mikep@microsoft.com), ~~Ram Jeyaraman (Ram.Jeyaraman@microsoft.com),~~ Microsoft

Editors:

Michael Pizzo (mikep@microsoft.com), Microsoft
Ralf Handl (ralf.handl@sap.com), SAP SE
Martin Zurmuehl (martin.zurmuehl@sap.com), SAP SE

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- *OData Version 4.01. Part 1: Protocol*. http://docs.oasis-open.org/odata/odata/v4.01/csd01/part1-protocol/odata-v4.01-csd01-part1-protocol.html.
- *OData Version 4.01. Part 2: URL Conventions* (this document). http://docs.oasis-open.org/odata/odata/v4.01/csd01/part2-url-conventions/odata-v4.01-csd01-part2-url-conventions.html.
- ~~*OData Version 4.01. Part 3: Common Schema Definition Language (CSDL). http://docs.oasis-open.org/odata/odata/v4.01/csd01/part3-csdl/odata-v4.01-csd01-part3-csdl.html.*~~
- *ABNF components: OData ABNF Construction Rules Version 4.01 and OData ABNF Test Cases*. ~~http://docs.oasis-open.org/odata/odata/v4.01/csd01/abnf/~~http://docs.oasis-open.org/odata/odata/v4.01/csprd02/abnf/.

Related work:

This specification replaces or supersedes:

- *OData Version 4.0 Part 2: URL Conventions*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. 24 February 2014. OASIS Standard. http://docs.oasis-open.org/odata/odata/v4.0/os/part2-url-conventions/odata-v4.0-os-part2-url-conventions.html. Latest version: http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html.

This specification is related to:

- *OData Vocabularies Version 4.0.* Edited by Mike Pizzo, Ralf Handl, and Ram Jeyaraman. Latest version: http://docs.oasis-open.org/odata/odata-vocabularies/v4.0/odata-vocabularies-v4.0.html.
- *OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01*, Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. Latest version: http://docs.oasis-open.org/odata/odata-csdl-json/v4.01/odata-csdl-json-v4.01.html.
- *OData Common Schema Definition Language (CSDL) XML Representation Version 4.01*, Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. Latest version: http://docs.oasis-open.org/odata/odata-csdl-xml/v4.01/odata-csdl-xml-v4.01.html.
- *OData JSON Format Version 4.01*. Edited by Ralf Handl, Michael Pizzo, and Mark Biamonte. Latest version: http://docs.oasis-open.org/odata/odata-json-format/v4.01/odata-json-format-v4.01.html.
- ~~*OData Atom Format Version 4.0*. Edited by Martin Zurmuehl, Michael Pizzo, and Ralf Handl. Latest version. http://docs.oasis-open.org/odata/odata-atom-format/v4.0/odata-atom-format-v4.0.html.~~

- *OData Extension for Data Aggregation Version 4.0*. Edited by Ralf Handl, Hubert Heijkers, Gerald Krause, Michael Pizzo, and Martin Zurmuehl. Latest version: http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/odata-data-aggregation-ext-v4.0.html.

Abstract:

The Open Data Protocol (OData) enables the creation of REST-based data services, which allow resources, identified using Uniform Resource Locators (URLs) and defined in a data model, to be published and edited by Web clients using simple HTTP messages. This specification defines a set of recommended (but not required) rules for constructing URLs to identify the data and metadata exposed by an OData service as well as a set of reserved URL query string operators.

Status:

This Working Draft (WD) has been produced by one or more TC Members; it has not yet been voted on by the TC or approved as a Committee Draft (Committee Specification Draft or a Committee Note Draft). The OASIS document Approval Process begins officially with a TC vote to approve a WD as a Committee Draft. A TC may approve a Working Draft, revise it, and re-approve it any number of times as a Committee Draft.

URI patterns:

Initial publication URI:
http://docs.oasis-open.org/odata/odata/v4.01/csd01/part2-url-conventions/odata-v4.01-csd01-part2-url-conventions.docx
Permanent “Latest version” URI:
http://docs.oasis-open.org/odata/odata/v4.01/os/part2-url-conventions/odata-v4.01-part2-url-conventions.docx

(Managed by OASIS TC Administration; please don’t modify.)

Table of Contents

1 Introduction

The Open Data Protocol (OData) enables the creation of REST-based data services, which allow resources, identified using Uniform Resource Locators (URLs) and defined in a data model, to be published and edited by Web clients using simple HTTP messages. This specification defines a set of recommended (but not required) rules for constructing URLs to identify the data and metadata exposed by an OData service as well as a set of reserved URL query string operators, which if accepted by an OData service, MUST be implemented as required by this document.

The ~~**[OData-Atom]** and~~ **[OData-JSON]** document~~s~~ specifies~~y~~ the format of the resource representations that are exchanged using OData and the **[OData-Protocol]** document describes the actions that can be performed on the URLs (optionally constructed following the conventions defined in this document) embedded in those representations.

Services are encouraged to follow the URL construction conventions defined in this specification when possible as consistency promotes an ecosystem of reusable client components and libraries.

1.1 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in **[RFC2119]**.

1.2 Normative References

[OData-ABNF]	*OData ABNF Construction Rules Version 4.01*. See the link in "Additional artifacts" section on cover page.
~~**[OData-Atom]**~~	~~*OData Atom Format Version 4.0*. See link in "Related work" section on cover page.~~
[OData-CSDLJSON]	*OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01~~ Part 3: Common Schema Definition Language (CSDL~~)*. See link in "~~Additional artifacts~~Related work" section on cover page.
[OData-CSDLXML]	*OData Common Schema Definition Language (CSDL) XML Representation Version 4.01.* See link in "Related work" section on cover page
[OData-JSON]	*OData JSON Format Version 4.01*. See link in "Related work" section on cover page.
[OData-Protocol]	*OData Version 4.01 Part 1: Protocol*. See link in "Additional artifacts" section on cover page.
[OData-VocCore]	*OData Vocabularies Version 4.0: Core Vocabulary.* See link in "Related work" section on cover page.
[RFC2119]	Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997. https://tools.ietf.org/html/rfc2119.
[RFC3986]	Berners-Lee, T., Fielding, R., and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax”, STD 66, RFC 3986, January 2005. https://tools.ietf.org/html/rfc3986.
[XML-Schema-2]	W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, D. Peterson, S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, P. V. Biron, A. Malhotra, Editors, W3C Recommendation, 5 April 2012, http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/. Latest version available at http://www.w3.org/TR/xmlschema11-2/.

1.3 Typographical Conventions

Keywords defined by this specification use this `monospaced` font.

```
Normative source code uses this paragraph style.
```

Some sections of this specification are illustrated with non-normative examples.

Example 1: text describing an example uses this paragraph style

```
Non-normative examples use this paragraph style.
```

All examples in this document are non-normative and informative only.

All other text is normative unless otherwise labeled.

2 URL Components

A URL used by an OData service has at most three significant parts: the *service root URL*, *resource path* and *query options*. Additional URL constructs (such as a fragment) can be present in a URL used by an OData service; however, this specification applies no further meaning to such additional constructs.

Example 2: OData URL broken down into its component parts:

```
http://host:port/path/SampleService.svc/Categories(1)/Products?$top=2&$orderby=Name
\______________________________________/\____________________/ \__________________/
                  |                               |                       |
          service root URL                  resource path           query options
```

Mandated and suggested content of these three significant URL components used by an OData service are covered in sequence in the three following chapters.

OData follows the URI syntax rules defined in **[RFC3986]** and in addition assigns special meaning to several of the sub-delimiters defined by **[RFC3986]**, so special care has to be taken regarding parsing and percent-decoding.

[RFC3986] defines three steps for URL processing that MUST be performed before percent-decoding:

- Split undecoded URL into components scheme, hier-part, query, and fragment at first "`:`", then first "`?`", and then first "`#`"
- Split undecoded hier-part into authority and path
- Split undecoded path into path segments at "`/`"

After applying these steps defined by RFC3986 the following steps MUST be performed:

- Split undecoded query at "`&`" into query options, and each query option at the first "`=`" into query option name and query option value
- Percent-decode path segments, query option names, and query option values exactly once
- Interpret path segments, query option names, and query option values according to OData rules

The OData rules are defined in this document and the **[OData-ABNF]**. Note that the ABNF is not expressive enough to define what a correct OData ~~URI~~ URL is in every imaginable use case. This specification document defines additional rules that a correct OData ~~URI~~ URL MUST fulfill. In case of doubt on what makes an OData ~~URI~~ URL correct the rules defined in this specification document take precedence. Note also that the rules in **[OData-ABNF]** assume that ~~URIs~~ URLs and ~~URI~~ URL parts have been percent-encoding normalized as described in section 6.2.2.2 of **[RFC3986]** before applying the grammar to them, i.e. all characters in the unreserved set (see rule `unreserved` in **[OData-ABNF]**) are plain literals and not percent-encoded. For characters outside of the unreserved set that are significant to OData the ABNF rules explicitly state whether the percent-encoded representation is treated identical to the plain literal representation. This is done to make the input strings in the ABNF test cases more readable.

~~One~~ For example, one of these rules is that single quotes within string literals are represented as two consecutive single quotes.

Example 3: valid OData URLs:

```
http://host/service/People('O''Neil')
http://host/service/People(%27O%27%27Neil%27)
http://host/service/People%28%27O%27%27Neil%27%29
http://host/service/Categories('Smartphone%2FTablet')
```

Example 4: invalid OData URLs:

```
http://host/service/People('O'Neil')
http://host/service/People('O%27Neil')
http://host/service/Categories('Smartphone/Tablet')
```

The first and second examples are invalid because a single quote in a string literal must be represented as two consecutive single quotes. The third example is invalid because forward slashes are interpreted as path segment separators and `Categories('Smartphone` is not a valid OData path segment, nor is `Tablet')`.

3 Service Root URL

The service root URL identifies the root of an OData service. A `GET` request to this URL returns the format-specific service document, see **[OData-JSON]**~~ and **[OData-Atom]**~~.

The service root URL ~~always terminates~~MUST terminate in a forward slash.

The service document enables simple hypermedia-driven clients to enumerate and explore the resources published by the OData service.

4 Resource Path

The rules for resource path construction as defined in this section are optional. OData services SHOULD follow the subsequently described URL path construction rules and are indeed encouraged to do so; as such consistency promotes a rich ecosystem of reusable client components and libraries.

Services that do not follow the resource path conventions for entity container children are strongly encouraged to document their resource paths by annotating entity container children with the term `Core.ResourcePath` defined in **[OData-VocCore]**. The annotation value is the URL of the annotated resource and may be relative to `xml:base` (if present), otherwise the request URL.

Resources exposed by an OData service are addressable by corresponding resource path URL components to enable interaction of the client with that resource aspect.

To illustrate the concept, some examples for resources might be: customers, a single customer, orders related to a single customer, and so forth. Examples of addressable aspects of these resources as exposed by the data model might be: collections of entities, a single entity, properties, links, operations, and so on.

An OData service MAY respond with `301 Moved Permanently` or `307 Temporary Redirect` from the canonical URL to the actual URL.

4.1 Addressing the Model for a Service

OData services expose their entity model according to **[OData-CSDLJSON]** or **[OData-CSDLXML]** at the metadata URL, formed by appending `$metadata` to the service root URL.

Example 5: Metadata document URL

```
http://host/service/$metadata
```

~~OData services MAY expose their entity model as a service, according to **[OData-CSDLXML]**, by appending a trailing slash (`/`) to the metadata document URL.~~

OData clients may request a particular format for metadata either through the Accept header or by using the `$format` system query option.

Example 6: Metadata ~~service~~ document~~root~~ URL with specified format

```
http://host/service/$metadata?$format=json~~/~~
```

4.2 Addressing the Batch Endpoint for a Service

OData services that support batch requests expose a batch URL formed by appending `$batch` to the service root URL.

Example 7: batch URL

```
http://host/service/$batch
```

4.3 Addressing Entities

The basic rules for addressing a collection (of entities), a single entity within a collection, a singleton, as well as a property of an entity are covered in the `resourcePath` syntax rule in **[OData-ABNF]**.

Below is a (non-normative) snippet from **[OData-ABNF]**:

```
resourcePath = entitySetName                  [collectionNavigation]
             / singleton                      [singleNavigation]
             / actionImportCall
             / entityColFunctionImportCall    [ collectionNavigation ]
             / entityFunctionImportCall       [ singleNavigation ]
             / complexColFunctionImportCall   [ collectionPath ]
             / complexFunctionImportCall      [ complexPath ]
             / primitiveColFunctionImportCall [ collectionPath ]
             / primitiveFunctionImportCall    [ singlePath ]
             / functionImportCallNoParens
             / crossjoin
             / '$all'                   [ "/" qualifiedEntityTypeName ]
```

Since OData has a uniform composable URL syntax and associated rules there are many ways to address a collection of entities, including, but not limited to:

- Via an entity set (see rule `entitySetName` in **[OData-ABNF]**)

  Example 8:

  ```
  http://host/service/Products
  ```

- By navigating a collection-valued navigation property (see rule: `entityColNavigationProperty`)
- By invoking a function that returns a collection of entities (see rule: `entityColFunctionCall`)

  Example 9: function with parameters in resource path

  ```
  http://host/service/ProductsByCategoryId(categoryId=2)
  ```

  Example 10: function with parameters as query options

  ```
  http://host/service/ProductsByColor(color=@color)?@color='red'
  ```

- By invoking an action that returns a collection of entities (see rule: `actionCall`)

Likewise there are many ways to address a single entity.

Sometimes a single entity can be accessed directly, for example by:

- Invoking a function that returns a single entity (see rule: `entityFunctionCall`)
- Invoking an action that returns a single entity (see rule: `actionCall`)
- Addressing a singleton

  Example 11:

  ```
  http://host/service/BestProductEverCreated
  ```

Often however a single entity is accessed by composing more path segments to a `resourcePath` that identifies a collection of entities, for example by:

- Using an entity key to select a single entity (see rules: `collectionNavigation` and `keyPredicate`)

  Example 12:

  ```
  http://host/service/Categories(1)
  ```

- Invoking an action bound to a collection of entities that returns a single entity (see rule: `boundOperation`)
- Invoking an function bound to a collection of entities that returns a single entity (see rule: `boundOperation`)

  Example 13:

  ```
  http://host/service/Products/Model.MostExpensive()
  ```

These rules are recursive, so it is possible to address a single entity via another single entity, a collection via a single entity and even a collection via a collection; examples include, but are not limited to:

- By following a navigation from a single entity to another related entity (see rule: `entityNavigationProperty`)

  Example 14:

  ```
  http://host/service/Products(1)/Supplier
  ```

- By invoking a function bound to a single entity that returns a single entity (see rule: `boundOperation`)

  Example 15:

  ```
  http://host/service/Products(1)/Model.MostRecentOrder()
  ```

- By invoking an action bound to a single entity that returns a single entity (see rule: `boundOperation`)
- By following a navigation from a single entity to a related collection of entities (see rule: `entityColNavigationProperty`)

  Example 16:

  ```
  http://host/service/Categories(1)/Products
  ```

- By invoking a function bound to a single entity that returns a collection of entities (see rule: `boundOperation`)

  Example 17:

  ```
  http://host/service/Categories(1)/Model.TopTenProducts()
  ```

- By invoking an action bound to a single entity that returns a collection of entities (see rule: `boundOperation`)
- By invoking a function bound to a collection of entities that returns a collection of entities (see rule: `boundOperation`)

  Example 18:

  ```
  http://host/service/Categories(1)/Products/Model.AllOrders()
  ```

- By invoking an action bound to a collection of entities that returns a collection of entities (see rule: `boundOperation`)

Finally it is possible to compose path segments onto a resource path that identifies a primitive, complex instance, collection of primitives or collection of complex instances and bind an action or function that returns an entity or collections of entities.

4.3.1 Canonical URL

For OData services conformant with the addressing conventions in this section, the canonical form of an absolute URL identifying a non-contained entity is formed by adding a single path segment to the service root URL. The path segment is made up of the name of the entity set associated with the entity followed by the key predicate identifying the entity within the collection. No type-cast segment is added to the canonical URL, even if the entity is an instance of a type derived from the declared entity type of its entity set.

The canonical key predicate for single-part keys consists only of the key property value without the key property name. For multi-part keys the key properties appear in the same order they appear in the key definition in the service metadata.

Example 19: Non-canonical URL

```
http://host/service/Categories(ID=1)/Products(ID=1)
```

Example 20: Canonical URL for previous example:

```
http://host/service/Products(1)
```

4.3.2 Canonical URL for Contained Entities

For contained entities (i.e. related via a containment navigation property~~ that specifies `ContainsTarget="true"`~~, see **[OData-CSDLJSON]** or **[OData-CSDLXML]**) the canonical URL is the canonical URL of the containing entity followed by:

- A cast segment if the navigation property is defined on a type derived from the entity type declared for the entity set,
- A path segment for the containment navigation property, and
- If the navigation property returns a collection, a key predicate that uniquely identifies the entity in that collection.

If the containment navigation property is defined on a complex type used within an ordered collection, the canonical URL of the containing entity is the canonical URL for the collection of complex types followed by a segment containing the zero-based ordinal of the complex typed instance within the collection.

4.3.3 URLs for Related Entities with Referential Constraints

If a navigation property leading to a related entity type has a partner navigation property that specifies a referential constraint, then those key properties of the related entity that take part in the referential constraint MAY be omitted from URLs.

Example 21: full key predicate of related entity

```
https://host/service/Orders(1)/Items(OrderID=1,ItemNo=2)
```

Example 22: shortened key predicate of related entity

```
https://host/service/Orders(1)/Items(2)
```

The two above examples are equivalent if the navigation property `Items` from `Order` to `OrderItem` has a partner navigation property from `OrderItem` to `Order` with a referential constraint tying the value of the `OrderID` key property of the `OrderItem` to the value of the `ID` key property of the `Order`.

The shorter form that does not specify the constrained key parts redundantly is preferred. If the value of the constrained key is redundantly specified, then it MUST match the principal key value.

4.3.4 Resolving an Entity-Id

To resolve an entity-id into a representation of the identified entity, the client issues a `GET` request to the `$entity` resource located at the URL `$entity` relative to the service root URL. The entity-id MUST be specified using the system query option `$id`. The entity-id may be expressed as an absolute IRI or relative to the ~~service~~ request root URL.

Example 23: request the entity representation for an entity-id

```
http://host/service/$entity?$id=Products(0)
```

The semantics of `$entity` are covered in the **[OData-Protocol]** document.

4.3.5 Alternate Keys

In addition to the canonical (primary) key an entity set or entity type can specify one or more alternate keys with the `Core.AlternateKeys` term (see **[OData-VocCore]**). Entities can be addressed via an alternate key using the same parentheses-style convention as for the canonical key, with one difference: single-part alternate keys MUST specify the key property name to unambiguously determine the alternate key.

Example 24: the same employee identified via the alternate key SSN, the canonical (primary) key ID using the non-canonical long form with specified key property name, and the canonical short form without key property name

```
http://host/service/Employees(SSN='123-45-6789')
http://host/service/Employees(ID='A1245')
http://host/service/Employees('A1245')
```

4.3.6 Key-as-Segment Convention

Services MAY support an alternate convention for addressing entities by appending a segment containing the unquoted key value to the URL of the collection containing the entity. Forward-slashes in key value segments MUST be percent-encoded; single quotes within key value segments are treated as part of the key value and do not need to be doubled or percent encoded.

Example 25: valid OData URLs:

```
http://host/service/Employees/A1245
http://host/service/People/O'Neil
http://host/service/People/O%27Neil
http://host/service/Categories/Smartphone%2FTablet
```

For multi-part keys, the entity MUST be addressed by multiple segments applied, one for each key value, in the order they appear in the metadata description of the entity key.

Example 26: multi-part key predicate, parentheses-style and key-as-segment

```
https://host/service/OrderItems(OrderID=1,ItemNo=2)
https://host/service/OrderItems/1/2
```

If a navigation property leading to a related entity type has a partner navigation property that specifies a referential constraint, then those key properties of the related entity that take part in the referential constraint MUST be omitted from URLs using key-as-segment convention.

Example 27: key predicate of related entity – no key segments for key properties of related entity with a referential constraint to preceding key segments

```
https://host/service/Orders/1/Items/2
```

The above example assumes that the navigation property `Items` from `Order` to `OrderItem` has a partner navigation property from `OrderItem` to `Order` with a referential constraint tying the value of the `OrderID` key property of the `OrderItem` to the value of the `ID` key property of the `Order`.

Because representing key values as segments could be ambiguous with other URL construction conventions, services that support key-as segment MUST implement the following precedence rules:

If a segment following an entity collection:

1) matches a defined OData segment (starting with “$”), treat it as such
2) matches a qualified bound function, bound action, or type name, treat it as such

3) matches an unqualified bound function, bound action, or type name defined in a default namespace (see **[OData-Protocol]**) treat it as such
4) treat as a key value

For maximum interoperability, services that support the key-as-segment convention SHOULD also support the canonical parentheses-style convention for addressing an entity within a collection, otherwise they MUST specify the URL for each returned entity in a response, as specified by the particular format.

Note: the key-as-segment convention can only be used with the canonical (primary) key and cannot be used with alternate keys as the key property names are not present in the keys and an alternative key cannot be determined.

4.4 Addressing References between Entities

OData services are based on a data model that supports relationships as first class constructs. For example, an OData service could expose a collection of Products entities each of which are related to a Category entity.

References between entities are addressable in OData just like entities themselves are (as described above) by appending a navigation property name followed by `/$ref` to the entity URL.

Example 28: URL addressing the references between `Categories(1)` and `Products`

```
http://host/service/Categories(1)/Products/$ref
```

Resource paths addressing a single entity reference can be used in `DELETE` requests to unrelated two entities. Resource paths addressing collection of references can be used in `DELETE` requests if they are followed by the system query option `$id` identifying one of the entity references in the collection. The entity-id specified by `$id` may be expressed absolute or relative to the request URL. For details see **[OData-Protocol]**.

Example 29: two ways of unrelating `Categories(1)` and `Products(0)`

```
DELETE http://host/service/Categories(1)/Products/$ref?$id=../../Products(0)
DELETE http://host/service/Products(0)/Category/$ref
```

4.5 Addressing Operations

The semantic rules for addressing and invoking actions and functions are defined in the **[OData-Protocol]** document.

Services MAY additionally support the use of the unqualified name of an action or function in a URL by defining one or more default namespaces through the `DefaultNamespace` term defined in **[OData-VocCore]**. For more information on default namespaces, see Default Namespaces in **[OData-Protocol]**.

4.5.1 Addressing Actions

~~The semantic rules for addressing and invoking actions are defined in the **[OData-Protocol]** document.~~
The grammar for addressing and invoking actions is defined by the following syntax grammar rules in **[OData-ABNF]**:

- The `actionImportCall` syntax rule defines the grammar in the `resourcePath` for addressing and invoking an action import directly from the service root.
- The `boundActionCall` syntax rule defines the grammar in the `resourcePath` for addressing and invoking an action that is appended to a `resourcePath` that identifies some resources that can be used as the binding parameter value when invoking the action.
- The `boundOperation` syntax rule (which encompasses the `boundActionCall` syntax rule), when used by the `resourcePath` syntax rule, illustrates how a `boundActionCall` can be appended to a `resourcePath`.
-

4.5.2 Addressing Functions

~~The semantic rules for addressing and invoking functions are defined in the **[OData-Protocol]** document.~~ The grammar for addressing and invoking functions is defined by a number syntax grammar rules in **[OData-ABNF]**, in particular:

- The function import call syntax rules `complexFunctionImportCall`, `complexColFunctionImportCall`, `entityFunctionImportCall`, `entityColFunctionImportCall`, `primitiveFunctionImportCall`, and `primitiveColFunctionImportCall` define the grammar in the `resourcePath` for addressing and providing parameters for a function import directly from the service root.
- The bound function call syntax rules `boundComplexFunctionCall`, `boundComplexColFunctionCall`, `boundEntityFunctionCall`, `boundEntityColFunctionCall`, `boundPrimitiveFunctionCall` and `boundPrimitiveColFunctionCall` define the grammar in the `resourcePath` for addressing and providing parameters for a function that is appended to a `resourcePath` that identifies some resources that can be used as the binding parameter value when invoking the function.
- The `boundOperation` syntax rule (which encompasses the bound function call syntax rules), when used by the `resourcePath` syntax rule, illustrates how a bound function call can be appended to a `resourcePath`.
- The `functionExpr` and `boundFunctionExpr` syntax rules as used by the `filter` and `orderby` syntax rules define the grammar for invoking functions to help filter and order resources identified by the `resourcePath` of the URL.
- The `aliasAndValue` syntax rule defines the grammar for providing function parameter values using Parameter Alias Syntax, see **[OData-Protocol]**.

4.6 Addressing a Property

To address an entity property clients append a path segment containing the property name to the URL of the entity. If the property has a complex type value, properties of that value can be addressed by further property name composition.

4.7 Addressing a Property Value

To address the raw value of a primitive property, clients append the~~a~~ path segment `/`~~containing the string~~ `$value` to the property URL.

Properties of type `Edm.Stream` already return the raw value of the media stream and do not support appending the `/$value` segment.

4.8 Addressing the Count of a Collection

To address the raw value of the number of items in a collection, clients append `/$count` to the resource path of the URL identifying the entity set or collection.

The `/$count` path suffix identifies the integer count of records in the collection and SHOULD NOT be combined with the system query options `$top`, `$skip`, `$orderby`, `$expand`, and `$format`. The count MUST NOT be affected by `$top`, `$skip`, `$orderby`, or `$expand`.

~~The `/$count` path suffix MAY be applied after `/$filter=` path segments, in which case the count is calculated after applying the filters.~~

~~-~~The ~~`/$count` path suffix MAY alternatively be combined with the `$filter` or `$search` system query options, in which case the~~ count is calculated after applying any `/$filter=` path segments, or ~~`$filter`~~`$filter` or ~~`$search`~~`$search` system query options to the collection. ~~The `/$count` path suffix MUST NOT be used in combination with both a `$filter=` path segment and a `$filter` or `$search` system query option.~~

~~Clients SHOULD NOT combine the system query options $top, $skip, $orderby, $expand, and $format with the path suffix /$count. The result of such a request is undefined.The returned count MUST NOT be affected by $top, $skip, $orderby, or $expand.~~

Example 30: the number of related entities

```
http://host/service/Categories(1)/Products/$count
```

Example 31: the number of entities in an entity set

```
http://host/service/Products/$count
```

Example 32: entity count in a `$filter` expression. Note that the spaces around `gt` are for readability of the example only; in real URLs they must be percent-encoded as `%20`.

```
http://host/service/Categories?$filter=Products/$count gt 0
```

Example 33: count of a filtered collection in a `$filter` expression; returns all Categories containing more than two products whose price is greater than 5.00.

```
http://host/service/Categories?
       $filter=Products/$count($filter=Price gt 5.00) gt 2
```

Example 30: entity count in an `$orderby` expression

```
http://host/service/Categories?$orderby=Products/$count
```

4.9 Addressing a Member within an Entity Collection

Collections of entities are modeled as entity sets, collection-valued navigation properties, or operation results.

For entity sets, results of operations associated with an entity set through an `EntitySet` or `EntitySetPath` declaration, or collection-valued navigation properties with a `NavigationPropertyBinding` or `ContainsTarget=true` specification, members of the collection can be addressed by convention by appending the parenthesized key ~~value~~ to the URL specifying the collection of entities~~. If the service supports~~, or by using the key-as-segment convention ~~, key segments can be appended instead of an parenthesized key~~if supported by the service.

4.10 Addressing a Member of an Ordered Collection

Collections can be annotated as ordered using the `Core.Ordered` term (see **[OData-VocCore]**). Individual items within an ordered collection of primitive or complex types can be addressed by appending a segment containing the zero-based ordinal to the URL of the collection. A negative ordinal number indexes from the end of the collection, with `-1` representing the last item in the collection. Services MUST NOT specify a negative index when returning the address of a member of an ordered collection.

Entity types are stably addressable using their canonical URL and SHOULD NOT be accessed or accessible using an index.

Example 31: the first address in a list of addresses for `MainSupplier`

```
http://host/service/MainSupplier/Addresses/0
```

4.11 Addressing Derived Types

Any resource path or path expression identifying a collection of entities or complex type instances can be appended with a path segment containing the qualified name of a type derived from the declared type of the collection. The result will be restricted to instances of the derived type and may be empty.

Any resource path or path expression identifying a single entity or complex type instance can be appended with a path segment containing the qualified name of a type derived from the declared type of the identified resource. If used in a resource path and the identified resource is not an instance of the derived type, the request will result in a `404 Not Found` response. If used in a path expression that is part of a Boolean expression, the type cast will evaluate to `null`.

Services MAY additionally support the use of the unqualified name of a derived type in a URL by defining one or more default namespaces through the `DefaultNamespace`~~ annotation~~ term defined in **[OData-VocCore]**. For more information on default namespaces, see Default Namespaces in **[OData-Protocol]**.

Services MAY also support treating an instance as a type outside of the type hierarchy using the same syntax and semantics as when addressing a derived type. In this case, the set and values of properties of the addressed type may be different than the properties of the source type. The set of such possible target types outside of the type hierarchy SHOULD be called out using the `Core.MayImplement` annotation term, defined in **[OData-VocCore]**.

Example 34: entity set restricted to `VipCustomer` instances

```
http://host/service/Customers/Model.VipCustomer
```

Example 35: entity restricted to a `VipCustomer` instance, resulting in `404 Not Found` if the customer with key `1` is not a `VipCustomer`

```
http://host/service/Customers/Model.VipCustomer(1)
http://host/service/Customers(1)/Model.VipCustomer
```

Example 36: cast the complex property `Address` to its derived type `DetailedAddress`, then get a property of the derived type

```
http://host/service/Customers(1)/Address/Model.DetailedAddress/Location
```

Example 37: filter expression with type cast; will evaluate to `null` for all non-`VipCustomer` instances and thus return only instances of `VipCustomer`

```
http://host/service/Customers?
       $filter=Model.VipCustomer/PercentageOfVipPromotionProductsOrdered gt 80
```

Example 38: expand the single related `Customer` only if it is an instance of `Model.VipCustomer`. For to-many relationships only `Model.VipCustomer` instances would be inlined,

```
http://host/service/Orders?$expand=Customer/Model.VipCustomer
```

4.12 Addressing a Subset of a Collection

Any resource path or path expression identifying a collection can be appended with a filter path segment~~,~~ ~~i.e /~~ consisting of `$filter=` followed by a parameter alias. A filter expression following the `filter` syntax rule in **[OData-ABNF]** MUST be assigned to the parameter alias in the query part of the request URL. The collection will be restricted to instances matching the filter expression assigned to the parameter alias and may be empty.

The `$filter=` path segment MAY occur multiple times and it MAY be combined with the `$filter` system query option.

Example 39: red products that cost less than 10 – combining path segment and system query option

```
GET Products/$filter=@foo?@foo=Price lt 10&$filter=Color eq 'red'
```

Example 40: red products that cost less than 10 – combine two path segments

```
GET Products/$filter=@p/$filter=c?@p=Price lt 10&@c=Color eq 'red'
```

Note that the `$filter=` path segment filters the "input" to the operation, and the `$filter` system query option filters the result of the operation, so the two can be used interchangeably only for `GET` operations. For a `PATCH` operation, the `$filter=` path segment is evaluated on the items *before* the modification and determines which items are to be modified, whereas the `$filter` system query option is evaluated *after* the modification and determines which of the modified items are part of the response.

The `$filter=` path segment MAY be followed by a path segment identifying a bound action or bound function applicable to the collection~~, and MAY be used in a set-based update or delete operation~~.

Example 41: invoke the `Special.Cluster` action on all products older than 3

```
POST /service/Products/$filter=@foo/Special.Cluster?@foo=Age gt 3
```

~~4.12~~4.13 Addressing Each Member of a Collection

To apply a `PATCH` or `DELETE` request to each member of a collection, clients append the path segment `/$each` to a resource path identifying a collection.

To apply a bound action or function ~~with a POST request~~ to each member of a collection, clients append the path segment `/$each` followed by a path segment identifying the bound action or function. The response is a collection of instances of the result type of the bound operation. If the bound operation returns a collection, the response is a collection of collections. System query options applied to the response can use `$it` to reference an item in the outermost collection, followed by `/any`, `/all`, or `/$count`.

The resource path of the collection preceding `/$each` MAY contain type-cast or filter segments to subset the collection.

~~4.13~~4.14 Addressing the Media Stream of a Media Entity

To address the media stream represented by a media entity, clients append `/$value` to the resource path of the media entity URL. Services may redirect from this canonical URL to the source URL of the media stream.

Example 42: request the media stream for the picture with the key value `Sunset4321299432`:

```
http://host/service/Pictures('Sunset4321299432')/$value
```

~~4.14~~4.15 Addressing the Cross Join of Entity Sets

In addition to querying related entities through navigation properties defined in the entity model of a service, the cross join operator allows querying across unrelated entity sets.

The cross join is addressed by appending the path segment `$crossjoin` to the service root URL, followed by the parenthesized comma-separated list of joined entity sets. It returns the Cartesian product of all the specified entity sets, represented as a collection of instances of a virtual complex type. Each instance consists of one non-nullable, single-valued navigation property per joined entity set. Each such navigation property is named identical to the corresponding entity set, with a target type equal to the declared entity type of the corresponding entity set.

The `$filter` and `$orderby` query options can be specified using properties of the entities in the selected entity sets, prepended with the entity set as the navigation property name.

The $expand query option can be specified using the names of the selected entity sets as navigation property names. If a selected entity set is not expanded, it MUST be represented using the read URL of the related entity as a navigation link in the complex type instance.

The $count, $skip, and $top query options can also be used with no special semantics.

Example 43: if `Sales` had a structural property `ProductID` instead of a navigation property `Product`, a "cross join" between `Sales` and `Products` could be addressed

```
http://host/service/$crossjoin(Products,Sales)?
                                        $filter=Products/ID eq Sales/ProductID
```

and would result in

```
{

"@odata.context":"http://host/service/$metadata#Collection(Edm.ComplexType)",
  "value":[
    {
      "Products@odata.navigationLink":"Products(0)",
      "Sales@odata.navigationLink":"Sales(42)",
    },
    {
      "Products@odata.navigationLink":"Products(0)",
      "Sales@odata.navigationLink":"Sales(57)",
    },
    ...
    {
      "Products@odata.navigationLink":"Products(99)",
      "Sales@odata.navigationLink":"Sales(21)",
    }
  ]
}
```

~~4.15~~4.16 Addressing All Entities in a Service

The symbolic resource `$all`, located at the service root, identifies the collection of all entities in a service, i.e. the union of all entity sets plus all singletons.

This symbolic resource is of type `Collection(Edm.EntityType)` and allows the $search system query option plus all other query options applicable to collections of entities.

The `$all` resource can be appended with a path segment containing the qualified name of an entity type in order to restrict the collections to entities of that type. Query options such as `$select`, `$filter`, `$expand` and `$orderby` can be applied to this restricted set according to the specified type.

Example 44: all entities in a service that somehow match `red`

```
http://host/service/$all?$search=red
```

Example 45: all `Customer` entities in a service whose name contains `red`

```
http://host/service/$all/Model.Customer?$filter=contains(Name,'red')
```

5 Query Options

The query options part of an OData URL specifies three types of information: system query options, custom query options, and parameter aliases. All OData services MUST follow the query string parsing and construction rules defined in this section and its subsections.

5.1 System Query Options

System query options are query string parameters that control the amount and order of the data returned for the resource identified by the URL. The names of all system query options are optionally prefixed with a dollar (`$`) character. 4.01 Services MUST support case-insensitive ~~-~~system query option names specified with or without the `$` prefix. Clients that want to work with 4.0 services MUST use lower case names and specify the `$` prefix.

For `GET`, `PATCH`, and `PUT` requests the following rules apply:

- Resource paths identifying a single entity, a complex type instance, a collection of entities, or a collection of complex type instances allow `$compute`, `$expand` and `$select`.
- Resource paths identifying a collection allow `$filter`, `$count`, `$orderby`, `$skip`, and `$top`.
- Resource paths identifying a collection of entities allow `$search`.
- Resource paths ending in `/$count` allow `$filter` and `$search`.
- Resource paths not ending in `/$count` or `/$batch` allow `$format`.

For `POST` requests to an action URL the return type of the action determines the applicable system query options that a service MAY support, following the same rules as `GET` requests.

`POST`~~, `PATCH`, and `PUT`~~ requests to an entity set ~~that return a sing~~follow the same rules as `GET` requests that return a single entity.

~~le instance of a structured type allow the system query options $compute, $expand and $select.~~

System query options SHOULD NOT be applied to a `DELETE` requests~~ do not return a result, so none of the system query options can be applied~~.

~~The semantics of system query options applied to `POST` requests to entity sets as well as all `PATCH`, `PUT` and `DELETE` requests are not defined by this specification and are reserved for future versions.~~

An OData service may support some or all of the system query options defined. If a data service does not support a system query option, it MUST reject any request that contains the unsupported option.

The same system query option~~-~~, irrespective of casing of whether or not it is prefixed with a `$`, MUST NOT be specified more than once for any resource.

The semantics of all system query options are defined in the **[OData-Protocol]** document.

The grammar and syntax rules for system query options are defined in **[OData-ABNF]**.

Dynamic properties can be used in the same way as declared properties. If they are not defined on an instance, they evaluate to `null`.

5.1.1 System Query Option `$filter`

The `$filter` system query option allows clients to filter a collection of resources that are addressed by a request URL. The expression specified with `$filter` is evaluated for each resource in the collection, and only items where the expression evaluates to true are included in the response. Resources for which the expression evaluates to false or to null, or which reference properties that are unavailable due to permissions, are omitted from the response.

The **[OData-ABNF]** `filter` syntax rule defines the formal grammar of the `$filter` query option.

5.1.1.1 Logical Operators

OData defines a set of logical operators that evaluate to true or false (i.e. a `boolCommonExpr` as defined in **[OData-ABNF]**). Logical operators are typically used to filter a collection of resources.

The syntax rules for the logical operators are defined in **[OData-ABNF]**. 4.01 Services MUST support case-insensitive operator names. Clients that want to work with 4.0 services MUST use lower case operator names.

The six comparison operators can be used with all primitive values except `Edm.Binary`, `Edm.Stream`, and the `Edm.Geo` types. `Edm.Binary`, `Edm.Stream`, and the `Edm.Geo` types can only be compared to the `null` value using the `eq` and `ne` operators.

When applied to operands of numeric types, numeric promotion rules are applied.

The `eq` and `in` operators can be used with collection-valued operands, and the `eq` operator can be used with structured operands.

5.1.1.1.1 Equals

The `eq` operator returns true if the left operand is equal to the right operand, otherwise it returns false.

When applied to operands of entity types, the `eq` operator returns true if both operands represent the same entity, or both operands represent null.

When applied to operands of complex types, the `eq` operator returns true if both operands have the same structure and same values, or both operands represent null.

When applied to ordered collections, the `eq` operator returns true if both operands have the same cardinality and each member of the left operand is equal to the corresponding member of the right operand.

For services that support comparing unordered collections, the `eq` operator returns true if both operands are equal after applying the same ordering on both collections~~contain the same members, in any order~~.

~~The~~ Each of the special values `null` `-INF`, and `INF` ~~value~~ is equal to itself, and only to itself.

The special value `NaN` is not equal to anything, even to itself.

5.1.1.1.2 Not Equals

The `ne` operator returns true if the left operand is not equal to the right operand, otherwise it returns false.

When applied to operands of entity types, the `ne` operator returns true if the two operands do not represent the same entity.

When applied to operands of complex types, the `ne` operator returns true if the operands do not have the same structure and same values.

When applied to ordered collections, the `ne` operator returns true if both operands do not have the same cardinality or any member of the left operand is not equal to the corresponding member of the right operand.

For services that support comparing unordered collections, the `ne` operator returns true if both operands do not have the same cardinality or do not contain the same members, in any order.

Each of the special values `null` `-INF`, and `INF` is not equal to any value but itself.

The special value `NaN` is not equal to anything, even to itself.

The `null` value is not equal to any value but itself.

5.1.1.1.3 Greater Than

The `gt` operator returns true if the left operand is greater than the right operand, otherwise it returns false.

The special value `INF` is greater than any number, date, or timestamp, and any number, date, or timestamp is greater than `-INF`.

~~For~~The B~~b~~oolean value~~s~~ `true` is greater than `false`.

If any operand is `null`, the operator returns false.

5.1.1.1.4 Greater Than or Equal

The `ge` operator returns true if the left operand is greater than or equal to the right operand, otherwise it returns false.

See rules for `gt` and `eq` for details.

5.1.1.1.5 Less Than

The `lt` operator returns true if the left operand is less than the right operand, otherwise it returns false.

The special value `-INF` is less than any number, date, or timestamp, and any number, date, or timestamp is less than `INF`.

The B~~b~~oolean value `false` is less than `true`.

If any operand is `null`, the operator returns false.

5.1.1.1.6 Less Than or Equal

The `le` operator returns true if the left operand is less than or equal to the right operand, otherwise it returns false.

See rules for `lt` and `eq` for details.

5.1.1.1.7 And

The `and` operator returns true if both the left and right operands evaluate to true, otherwise it returns false.

The `null` value is treated as unknown, so if one operand evaluates to `null` and the other operand to false, the `and` operator returns false. All other combinations with `null` return `null`.

5.1.1.1.8 Or

The `or` operator returns false if both the left and right operands both evaluate to false, otherwise it returns true.

The `null` value is treated as unknown, so if one operand evaluates to `null` and the other operand to true, the `or` operator returns true. All other combinations with `null` return `null`.

5.1.1.1.9 Not

The `not` operator returns true if the operand returns false, otherwise it returns false.

The `null` value is treated as unknown, so `not null` returns `null`.

5.1.1.1.10 Has

The `has` operator returns true if the right ~~hand~~ operand is an enumeration value whose flag(s) are set on the left operand.

The `null` value is treated as unknown, so if one operand evaluates to `null`, the `has` operator returns `null`.

5.1.1.1.11 In

The `in` operator returns true if the left ~~hand~~ operand is a member of the right ~~hand~~ operand. The right ~~hand~~ operand MUST be either a comma-separated list of primitive values, enclosed in parentheses, or a single expression that resolves to a collection ~~of the appropriate type~~.

5.1.1.1.12 Logical Operator Examples

The following examples illustrate the use and semantics of each of the logical operators.

Example 46: all products with a `Name` equal to '`Milk`'

```
http://host/service/Products?$filter=Name eq 'Milk'
```

Example 47: all products with a `Name` not equal to '`Milk`'

```
http://host/service/Products?$filter=Name ne 'Milk'
```

Example 48: all products with a Name greater than 'Milk':

```
http://host/service/Products?$filter=Name gt 'Milk'
```

Example 49: all products with a Name greater than or equal to 'Milk':

```
http://host/service/Products?$filter=Name ge 'Milk'
```

Example 50: all products with a Name less than 'Milk':

```
http://host/service/Products?$filter=Name lt 'Milk'
```

Example 51: all products with a Name less than or equal to 'Milk':

```
http://host/service/Products?$filter=Name le 'Milk'
```

Example 52: all products with the Name 'Milk' that also have a Price less than 2.55:

```
http://host/service/Products?$filter=Name eq 'Milk' and Price lt 2.55
```

Example 53: all products that either have the Name 'Milk' or have a Price less than 2.55:

```
http://host/service/Products?$filter=Name eq 'Milk' or Price lt 2.55
```

Example 54: all products that do not have a Name that ends with 'ilk':

```
http://host/service/Products?$filter=not endswith(Name,'ilk')
```

Example 55: all products whose `style` value includes `Yellow`:

```
http://host/service/Products?$filter=style has Sales.Pattern'Yellow'
```

Example 56: all products whose `name` value is ‘Milk’ or ‘Cheese’:

```
http://host/service/Products?$filter=Name in ('Milk', 'Cheese')
```

5.1.1.2 Arithmetic Operators

OData defines a set of arithmetic operators that require operands that evaluate to numeric types. Arithmetic operators are typically used to filter a collection of resources. However, services MAY allow using arithmetic operators with the `$orderby` system query option.

If an operand of an arithmetic operator is null, the result is null.

The syntax rules for the arithmetic operators are defined in **[OData-ABNF]**. 4.01 Services MUST support case-insensitive operator names. Clients that want to work with 4.0 services MUST use lower case operator names.

5.1.1.2.1 Addition

The `add` operator adds the left and right numeric operands.

For operands of type `Edm.Decimal` the scale of the result is scaleof(A add B) = max(scaleof(A), scaleof(B)), or variable if any operand has variable scale.

The `add` operator is also valid for the following time-related operands:

- `DateTimeOffset add Duration` results in a `DateTimeOffset`
- `Duration add Duration` results in a `Duration`
- `Date add Duration` results in a `Date`

The rules for time-related operands are defined in **[XML-Schema-2]**, section E.3.3. Specifically, for adding a duration to a date:

- Convert date to datetime (in any timezone) with a zero time component
- Add/subtract duration
- Convert to date by removing the time and timezone components

Thus today plus a positive duration smaller than one day is today; today minus a positive duration smaller than one day is yesterday.

5.1.1.2.2 Subtraction

The `sub` operator subtracts the right numeric operand from the left numeric operand.

For operands of type `Edm.Decimal` the scale of the result is scaleof(A sub B) = max(scaleof(A), scaleof(B)), or variable if any operand has variable scale.

The `sub` operator is also valid for the following time-related operands:

- `DateTimeOffset sub Duration` results in a `DateTimeOffset`
- `Duration sub Duration` results in a `Duration`
- `DateTimeOffset sub DateTimeOffset` results in a `Duration`
- `Date sub Duration` results in a `Date`
- `Date sub Date` results in a `Duration`

The rules for time-related operands are defined in **[XML-Schema-2]**, section E.3.3. Specifically for subtracting a duration from a date see the preceding section.

5.1.1.2.3 Negation

The negation operator, represented by a minus (-) sign, changes the sign of its numeric or `Duration` operand.

5.1.1.2.4 Multiplication

The `mul` operator multiplies the left and right numeric operands. The `mul` operator is also valid for multiplying a `Duration` value with a numeric value.

For operands of type `Edm.Decimal` the scale of the result is scaleof(A mul B) = scaleof(A) + scaleof(B), or variable if any operand has variable scale.

5.1.1.2.5 Division

The `div` and `divby` operators divide the left numeric operand by the right numeric operand. They are also valid for dividing a `Duration` value by a numeric value.

~~If the right operand is zero and the left operand is neither of type `Edm.Single` nor `Edm.Double`, the request fails.~~ If the left operand is numeric~~of type `Edm.Single` or `Edm.Double`~~, then positive `div` zero returns `INF`, negative `div` zero returns `-INF`, and zero `div` zero returns `NaN`. For 4.0 services this ~~special treatment~~behavior is only required if~~If the right operand is zero and~~ the left operand is ~~neither~~ of type `Edm.Single` ~~n~~or `Edm.Double`, for all other types the request MAY fail~~s~~.

For operands of type `Edm.Decimal` the result is computed with maximal decimal scale. If any operand has variable scale, the result has variable scale. Otherwise the resulting scale is service-specific, and clients can use `cast` to force the result to a specific scale.

The `div` and `divby` operators differ in their handling of integers. If both operands to `div` are of an integer type, the result is an integer representing the whole number of times the right operator fits into the left operator. The `divby` operator, on the other hand, promotes both operands to decimal before computing the operation, and may yield a fractional result.

5.1.1.2.6 Modulo

The `mod` operator returns the remainder when the left numeric operand is divided by the right numeric operand. The sign of the result is the same as the sign of the left operand. If the right operand is zero, the request fails.

For operands of type `Edm.Decimal` the scale of the result is scaleof(A mod B) = max(scaleof(A), scaleof(B)), or variable if any operand has variable scale.

5.1.1.2.7 Arithmetic Operator Examples

The following examples illustrate the use and semantics of each of the Arithmetic operators.

Example 57: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price add 2.45 eq 5.00
```

Example 58: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price sub 0.55 eq 2.00
```

Example 59: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price mul 2.0 eq 5.10
```

Example 60: all products with a Price of 2.55:

```
http://host/service/Products?$filter=Price div 2.55 eq 1
```

Example 61: all products with an integer Rating value of 4 or 5:

```
http://host/service/Products?$filter=Rating div 2 eq 2
```

Example 62: all products with an integer Rating value of 5:

```
http://host/service/Products?$filter=Rating divby 2 eq 2.5
```

Example 63: all products with a Rating exactly divisible by 5:

```
http://host/service/Products?$filter=Rating mod 5 eq 0
```

5.1.1.3 Grouping

The Grouping operator (open and close parenthesis "`( )`") controls the evaluation order of an expression. The Grouping operator returns the expression grouped inside the parenthesis.

Example 64: all products because 9 mod 3 is 0

```
http://host/service/Products?$filter=(4 add 5) mod (4 sub 1) eq 0
```

5.1.1.4 Canonical Functions

In addition to operators, a set of functions is also defined for use with the `$filter` or `$orderby` system query options. The following sections describe the available functions. Note: ISNULL or COALESCE operators are not defined. Instead, OData defines a `null` literal that can be used in comparisons.

If a parameter of a canonical function is `null`, the function returns `null`.

The syntax rules for all functions are defined in **[OData-ABNF]**. 4.01 Services MUST support case-insensitive canonical function names. Clients that want to work with 4.0 services MUST use lower case canonical function names.

5.1.1.5 String and Collection Functions

5.1.1.5.1 `concat`

The `concat` function has three overloads, with the following signatures:

```
Edm.String concat(Edm.String,Edm.String)
Collection concat(Collection,Collection)
OrderedCollection concat(OrderedCollection,OrderedCollection)
```

The `concat` function with string parameter values returns a string that appends the second ~~input~~ parameter string value to the first.

The concat function with collection parameter values returns a collection that appends all items of the second collection to the first. If both collections are ordered, the result is also ordered.

The `concatMethodCallExpr` syntax rule defines how the `concat` function is invoked.

Example 65: all customers from Berlin, Germany

```
http://host/service/Customers?
                  $filter=concat(concat(City,', '), Country) eq 'Berlin, Germany'
```

5.1.1.5.2 `contains`

The `contains` function has two overloads, with the following signatures:

```
Edm.Boolean contains(Edm.String,Edm.String)

Edm.Boolean contains(OrderedCollection,OrderedCollection)
```

The `contains` function with string parameter values returns true if the second ~~parameter~~ string ~~value~~ is a substring of the first ~~parameter~~ string ~~value~~, otherwise it returns false.

The `contains` function with ordered collection parameter values returns true if the first collection can be transformed into the second collection by removing zero or more items from the beginning or the end of the first collection.

The `containsMethodCallExpr` syntax rule defines how the `contains` function is invoked.

Example 66: all customers with a `CompanyName` that contains `'Alfreds'`

```
http://host/service/Customers?$filter=contains(CompanyName,'Alfreds')
```

5.1.1.5.3 `endswith`

The `endswith` function has two overloads, with the following signatures:

```
Edm.Boolean endswith(Edm.String,Edm.String)
Edm.Boolean endswith(OrderedCollection,OrderedCollection)
```

The `endswith` function with string parameter values returns true if the first ~~parameter~~ string ~~value~~ ends with the second ~~parameter~~ string ~~value~~, otherwise it returns false.

The `endswith` function with ordered collection parameter values returns true if the first collection can be transformed into the second collection by removing zero or more items from the beginning of the first collection.

The `endsWithMethodCallExpr` syntax rule defines how the `endswith` function is invoked.

Example 67: all customers with a `CompanyName` that ends with `'Futterkiste'`

```
http://host/service/Customers?$filter=endswith(CompanyName,'Futterkiste')
```

5.1.1.5.4 `indexof`

The `indexof` function has two overloads, with the following signatures:

```
Edm.Int32 indexof(Edm.String,Edm.String)
Edm.Int32 indexof(OrderedCollection,OrderedCollection)
```

The `indexof` function with string parameter values returns the zero-based character position of the first occurrence of the second ~~parameter~~ string~~value~~ in the first ~~parameter~~ string~~value~~, or -1 if the ~~second~~ first ~~parameter~~ string~~value~~ does not ~~occur~~ contain ~~in~~ the second ~~first parameter~~ string~~value~~.

-The `indexof` function with ordered collection parameter values returns the zero-based index of the first occurrence of the second collection in the first collection, or -1 if the first collection does not contain the second collection.

The `indexOfMethodCallExpr` syntax rule defines how the `indexof` function is invoked.

Example 68: all customers with a `CompanyName` containing `'lfreds'` starting at the second character

```
http://host/service/Customers?$filter=indexof(CompanyName,'lfreds') eq 1
```

5.1.1.5.5 `length`

The `length` function has two overloads, with the following signatures:

```
Edm.Int32 length(Edm.String)
Edm.Int32 length(Collection)
```

The `length` function with a string parameter value returns the number of characters in the string ~~parameter value~~.

The `length` function with a collection parameter value returns the number of itens in the collection.

The `lengthMethodCallExpr` syntax rule defines how the `length` function is invoked.

Example 69: all customers with a `CompanyName` that is 19 characters long

```
http://host/service/Customers?$filter=length(CompanyName) eq 19
```

5.1.1.5.6 `startswith`

The `startswith` function has two overloads, with the following signatures:

```
Edm.Boolean startswith(Edm.String,Edm.String)
Edm.Boolean startswith(Collection,Collection)
```

The `startswith` function with string parameter values returns true if the first ~~parameter~~ string ~~value~~ starts with the second ~~parameter~~ string ~~value~~, otherwise it returns false.

The `startswith` function with ordered collection parameter values returns true if the first collection can be transformed into the second collection by removing zero or more items from the end of the first collection.

The `startsWithMethodCallExpr` syntax rule defines how the `startswith` function is invoked.

Example 70: all customers with a `CompanyName` that starts with `'Alfr'`

```
http://host/service/Customers?$filter=startswith(CompanyName,'Alfr')
```

5.1.1.5.7 `substring`

The `substring` function has f~~consists of two~~our overloads, with the following signatures:

```
Edm.String substring(Edm.String,Edm.Int32)
Edm.String substring(Edm.String,Edm.Int32,Edm.Int32)
OrderedCollection substring(OrderedCollection,Edm.Int32)
OrderedCollection substring(OrderedCollection,Edm.Int32,Edm.Int32)
```

The two-~~-~~argument `substring` function with string parameter values returns a substring of the first parameter string value, starting at the Nth character and finishing at the last character (where N is the second parameter integer value). The three-~~-~~argument `substring` function with string parameter values returns a substring of the first parameter string value identified by selecting up to M characters starting at the Nth character (where N is the second parameter integer value and M is the third parameter integer value).

The two-argument `substring` function with ordered collection parameter values returns an ordered collection consisting of all items of the first collection starting at the Nth item and finishing at the last item. The three-argument `substring` function with ordered collection parameter values returns an ordered collection consisting of up to M items of the first collection starting at the Nth item (where N is the second parameter integer value and M is the third parameter integer value).

The start index N is zero-based.

If the start index N is larger than the length of the string/collection, an empty string/collection is returned.

If the length M is larger than the length of the remaining string/collection starting at the Nth character/item, as many characters/items as are available are returned.

A negative length M is a bad request.

A negative start index N, if supported, returns a string/collection starting N characters/items before the end of the string/collection.

The `substringMethodCallExpr` syntax rule defines how the `substring` function is ~~are~~ invoked.

Example 71: all customers with a `CompanyName` of `'lfreds Futterkiste'` once the first character has been removed

```
http://host/service/Customers?
                      $filter=substring(CompanyName, 1) eq 'lfreds Futterkiste'
```

Example 72: all customers with a `CompanyName` that has `'lf'` as the second and third characters, e.g, `'Alfreds Futterkiste'`

```
http://host/service/Customers?$filter=substring(CompanyName,1,2) eq 'lf'
```

5.1.1.6 Collection Functions

5.1.1.6.1 `hassubset`

The `hassubset` function has the following signature:

`Edm.Boolean hassubset(Collection, Collection)`

The `hassubset` function returns true if the first collection can be transformed into the second collection by reordering and/or removing zero or more items. The `hasSubsetMethodCallExpr` syntax rule defines how the `hassubset` function is invoked.

Example 73: `hassubset` expressions that return true

```
hassubset([4,1,3],[4,1,3])
hassubset([4,1,3],[1,3,4])
hassubset([4,1,3],[3,1])
hassubset([4,1,3],[4,3])
hassubset([4,1,3,1],[1,1])
```

Example 74: `hassubset` expression that returns false: 1 appears only once in the left operand

```
hassubset([1,2],[1,1,2])
```

5.1.1.6.2 `hassubsequence`

The `hassubsequence` function has the following signature:

`Edm.Boolean hassubsequence(OrderedCollection,OrderedCollection)`

The `hassubsequence` function returns true if the first collection can be transformed into the second collection by removing zero or more items. The `hasSubsequenceMethodCallExpr` syntax rule defines how the `hassubsequence` function is invoked.

Example 75: `hassubsequence` expressions that return true

```
hassubsequence([4,1,3],[4,1,3])
hassubsequence([4,1,3],[4,1])
hassubsequence([4,1,3],[4,3])
hassubsequence([4,1,3,1],[1,1])
```

Example 76: `hassubsequence` expressions that return false

```
hassubsequence([4,1,3],[1,3,4])
hassubsequence([4,1,3],[3,1])
hassubsequence([1,2],[1,1,2])
```

5.1.1.7 String Functions

~~5.1.1.5.8~~5.1.1.7.1 `tolower`

The `tolower` function has the following signature:

```
Edm.String tolower(Edm.String)
```

The `tolower` function returns the input parameter string value with all uppercase characters converted to lowercase according to Unicode rules. The `toLowerMethodCallExpr` syntax rule defines how the `tolower` function is invoked.

Example 77: all customers with a `CompanyName` that equals `'alfreds futterkiste'` once any uppercase characters have been converted to lowercase

```
http://host/service/Customers?
                          $filter=tolower(CompanyName) eq 'alfreds futterkiste'
```

~~5.1.1.5.9~~5.1.1.7.2 `toupper`

The `toupper` function has the following signature:

```
Edm.String toupper(Edm.String)
```

The `toupper` function returns the input parameter string value with all lowercase characters converted to uppercase according to Unicode rules. The `toUpperMethodCallExpr` syntax rule defines how the `toupper` function is invoked.

Example 78: all customers with a `CompanyName` that equals `'ALFREDS FUTTERKISTE'` once any lowercase characters have been converted to uppercase

```
http://host/service/Customers?
                          $filter=toupper(CompanyName) eq 'ALFREDS FUTTERKISTE'
```

~~5.1.1.5.10~~5.1.1.7.3 `trim`

The `trim` function has the following signature:

```
Edm.String trim(Edm.String)
```

The `trim` function returns the input parameter string value with all leading and trailing whitespace characters, according to Unicode rules, removed. The `trimMethodCallExpr` syntax rule defines how the `trim` function is invoked.

Example 79: all customers with a `CompanyName` without leading or trailing whitespace characters

```
http://host/service/Customers?$filter=trim(CompanyName) eq CompanyName
```

~~5.1.1.6~~5.1.1.8 Date and Time Functions

~~5.1.1.6.1~~5.1.1.8.1 date

The `date` function has the following signature:

```
Edm.Date date(Edm.DateTimeOffset)
```

The `date` function returns the date part of the `DateTimeOffset` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value.

~~5.1.1.6.2~~5.1.1.8.2 day

The `day` function has the following signatures:

```
Edm.Int32 day(Edm.Date)
Edm.Int32 day(Edm.DateTimeOffset)
```

The `day` function returns the day component `Date` or `DateTimeOffset` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value. The `dayMethodCallExpr` syntax rule defines how the `day` function is invoked.

Services that are unable to preserve the offset of `Edm.DateTimeOffset` values and instead normalize the values to some common time zone (i.e. UTC) MUST fail evaluation of the `day` function for literal `Edm.DateTimeOffset` values that are not stated in the time zone of the normalized values.

Example 80: all employees born on the 8th day of a month

```
http://host/service/Employees?$filter=day(BirthDate) eq 8
```

~~5.1.1.6.3~~5.1.1.8.3 fractionalseconds

The `fractionalseconds` function has the following signatures:

```
Edm.Decimal fractionalseconds(Edm.DateTimeOffset)
Edm.Decimal fractionalseconds(Edm.TimeOfDay)
```

The `fractionalseconds` function returns the fractional seconds component of the `DateTimeOffset` or `TimeOfDay` parameter value as a non-negative decimal value less than 1. The `fractionalsecondsMethodCallExpr` syntax rule defines how the `fractionalseconds` function is invoked.

Example 81: all employees born less than 100 milliseconds after a full second of any minute of any hour on any day

```
http://host/service/Employees?$filter=fractionalseconds(BirthDate) lt 0.1
```

~~5.1.1.6.4~~5.1.1.8.4 hour

The `hour` function has the following signatures:

```
Edm.Int32 hour(Edm.DateTimeOffset)
Edm.Int32 hour(Edm.TimeOfDay)
```

The `hour` function returns the hour component of the `DateTimeOffset` or `TimeOfDay` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value. The `hourMethodCallExpr` syntax rule defines how the `hour` function is invoked.

Services that are unable to preserve the offset of `Edm.DateTimeOffset` values and instead normalize the values to some common time zone (i.e. UTC) MUST fail evaluation of the `hour` function for literal `Edm.DateTimeOffset` values that are not stated in the time zone of the normalized values.

Example 82: all employees born in the 4th hour of a day

```
http://host/service/Employees?$filter=hour(BirthDate) eq 4
```

~~5.1.1.6.5~~5.1.1.8.5 maxdatetime

The `maxdatetime` function has the following signature:

```
Edm.DateTimeOffset maxdatetime()
```

The `maxdatetime` function returns the latest possible point in time as a `DateTimeOffset` value.

~~5.1.1.6.6~~5.1.1.8.6 mindatetime

The `mindatetime` function has the following signature:

```
Edm.DateTimeOffset mindatetime()
```

The `mindatetime` function returns the earliest possible point in time as a `DateTimeOffset` value.

~~5.1.1.6.7~~5.1.1.8.7 minute

The `minute` function has the following signatures:

```
Edm.Int32 minute(Edm.DateTimeOffset)
Edm.Int32 minute(Edm.TimeOfDay)
```

The `minute` function returns the minute component of the `DateTimeOffset` or `TimeOfDay` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value. The `minuteMethodCallExpr` syntax rule defines how the `minute` function is invoked.

Example 83: all employees born in the 40th minute of any hour on any day

```
http://host/service/Employees?$filter=minute(BirthDate) eq 40
```

~~5.1.1.6.8~~5.1.1.8.8 month

The `month` function has the following signatures:

```
Edm.Int32 month(Edm.Date)
Edm.Int32 month(Edm.DateTimeOffset)
```

The `month` function returns the month component of the `Date` or `DateTimeOffset` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value. The `monthMethodCallExpr` syntax rule defines how the `month` function is invoked.

Services that are unable to preserve the offset of `Edm.DateTimeOffset` values and instead normalize the values to some common time zone (i.e. UTC) MUST fail evaluation of the `month` function for literal `Edm.DateTimeOffset` values that are not stated in the time zone of the normalized values.

Example 84: all employees born in May

```
http://host/service/Employees?$filter=month(BirthDate) eq 5
```

~~5.1.1.6.9~~5.1.1.8.9 now

The `now` function has the following signature:

```
Edm.DateTimeOffset now()
```

The `now` function returns the current point in time (date and time with time zone) as a `DateTimeOffset` value.

Services are free to choose the time zone for the current point, e.g. UTC. Services that are unable to preserve the offset of `Edm.DateTimeOffset` values and instead normalize the values to some common time zone SHOULD return a value in the normalized time zone (i.e., UTC).

~~5.1.1.6.10~~5.1.1.8.10 second

The `second` function has the following signatures:

```
Edm.Int32 second(Edm.DateTimeOffset)
Edm.Int32 second(Edm.TimeOfDay)
```

The `second` function returns the second component (without the fractional part) of the `DateTimeOffset` or `TimeOfDay` parameter value. The `secondMethodCallExpr` syntax rule defines how the `second` function is invoked.

Example 85: all employees born in the 40th second of any minute of any hour on any day

```
http://host/service/Employees?$filter=second(BirthDate) eq 40
```

~~5.1.1.6.11~~5.1.1.8.11 time

The `time` function has the following signature:

```
Edm.TimeOfDay time(Edm.DateTimeOffset)
```

The `time` function returns the time part of the `DateTimeOffset` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value.

Services that are unable to preserve the offset of `Edm.DateTimeOffset` values and instead normalize the values to some common time zone (i.e. UTC) MUST fail evaluation of the `time` function for literal `Edm.DateTimeOffset` values that are not stated in the time zone of the normalized values.

~~5.1.1.6.12~~5.1.1.8.12 totaloffsetminutes

The `totaloffsetminutes` function has the following signature:

```
Edm.Int32 totaloffsetminutes(Edm.DateTimeOffset)
```

The `totaloffsetminutes` function returns the signed number of minutes in the time zone offset part of the `DateTimeOffset` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value.

~~5.1.1.6.13~~5.1.1.8.13 totalseconds

The `totalseconds` function has the following signature:

```
Edm.Decimal totalseconds(Edm.Duration)
```

The `totalseconds` function returns the duration of the value in total seconds, including fractional seconds.

~~5.1.1.6.14~~5.1.1.8.14 year

The `year` function has the following signatures:

```
Edm.Int32 year(Edm.Date)
Edm.Int32 year(Edm.DateTimeOffset)
```

The `year` function returns the year component of the `Date` or `DateTimeOffset` parameter value, evaluated in the time zone of the `DateTimeOffset` parameter value. The `yearMethodCallExpr` syntax rule defines how the `year` function is invoked.

Services that are unable to preserve the offset of `Edm.DateTimeOffset` values and instead normalize the values to some common time zone (i.e. UTC) MUST fail evaluation of the `year` function for literal `Edm.DateTimeOffset` values that are not stated in the time zone of the normalized values.

Example 86: all employees born in 1971

```
http://host/service/Employees?$filter=year(BirthDate) eq 1971
```

~~5.1.1.7~~5.1.1.9 Arithmetic Functions

~~5.1.1.7.1~~5.1.1.9.1 `ceiling`

The `ceiling` function has the following signatures

```
Edm.Double ceiling(Edm.Double)
Edm.Decimal ceiling(Edm.Decimal)
```

The `ceiling` function rounds the input numeric parameter up to the nearest numeric value with no decimal component. The `ceilingMethodCallExpr` syntax rule defines how the `ceiling` function is invoked.

Example 87: all orders with freight costs that round up to 32

```
http://host/service/Orders?$filter=ceiling(Freight) eq 32
```

~~5.1.1.7.2~~5.1.1.9.2 `floor`

The `floor` function has the following signatures

```
Edm.Double floor(Edm.Double)
Edm.Decimal floor(Edm.Decimal)
```

The `floor` function rounds the input numeric parameter down to the nearest numeric value with no decimal component. The `floorMethodCallExpr` syntax rule defines how the `floor` function is invoked.

Example 88: all orders with freight costs that round down to 32

```
http://host/service/Orders?$filter=floor(Freight) eq 32
```

~~5.1.1.7.3~~5.1.1.9.3 `round`

The `round` function has the following signatures

```
Edm.Double round(Edm.Double)
Edm.Decimal round(Edm.Decimal)
```

The `round` function rounds the input numeric parameter to the nearest numeric value with no decimal component. The mid-point between two integers is rounded away from zero, i.e. 0.5 is rounded to 1 and -0.5 is rounded to -1. The `roundMethodCallExpr` syntax rule defines how the `round` function is invoked.

Example 89: all orders with freight costs that round to 32

```
http://host/service/Orders?$filter=round(Freight) eq 32
```

~~5.1.1.8~~5.1.1.10 Type Functions

~~5.1.1.8.1~~5.1.1.10.1 `cast`

The `cast` function has the following signatures:

```
type cast(type)
type cast(expression,type)
```

The single parameter `cast` function returns the current instance cast to the type specified. The two-parameter `cast` function returns the object referred to by the expression cast to the type specified.

The `cast` function follows these assignment rules:

- The `null` value can be cast to any type.
- Primitive types are cast to `Edm.String` or a type definition based on it by using the literal representation used in payloads, and WKT (well-known text) format for `Geo` types, see rules `fullCollectionLiteral`, `fullLineStringLiteral`, `fullMultiPointLiteral`, `fullMultiLineStringLiteral`, `fullMultiPolygonLiteral`, `fullPointLiteral`, and `fullPolygonLiteral` in **[OData-ABNF]**. The cast fails if the target type specifies an insufficient `MaxLength`.
- `Edm.String`, or a type definition based on `Edm.String`, can be cast to a primitive type if the string contains a literal representation for the target type.
- Numeric primitive types are cast to each other with appropriate rounding. The cast fails if the integer part doesn't fit into target type.
- `Edm.DateTimeOffset`, `Edm.Duration`, and `Edm.TimeOfDay` values can be cast to the same type with a different precision with appropriate rounding.
- Non-Unicode string values can be cast to a Unicode string type definition. Unicode string values can be cast to non-Unicode string type definition if the Unicode string only contains ASCII characters.
- Structured types are assignable to their type or a direct or indirect base type.
- Collections are cast item by item.
- ~~Strings with~~

The `cast` function is optional for primitive values (first five rules) and up-casts (~~sixth~~ seventh rule).

If the cast fails, the `cast` function returns `null`.

~~5.1.1.8.2~~5.1.1.10.2 `isof`

The `isof` function has the following signatures

```
Edm.Boolean isof(type)
Edm.Boolean isof(expression,type)
```

The single parameter `isof` function returns true if the current instance is assignable to the type specified, according to the assignment rules for the `cast` function, otherwise it returns `false`.

The two parameter `isof` function returns true if the object referred to by the expression is assignable to the type specified, according to the same rules, otherwise it returns false.

The `isofExpr` syntax rule defines how the `isof` function is invoked.

Example 90: orders that are also `BigOrders`

```
http://host/service/Orders?$filter=isof(NorthwindModel.BigOrder)
http://host/service/Orders?$filter=isof($it,NorthwindModel.BigOrder)
```

Example 91: orders of a customer that is a `VIPCustomer`

```
http://host/service/Orders?$filter=isof(Customer,NorthwindModel.VIPCustomer)
```

~~5.1.1.9~~5.1.1.11 Geo Functions

~~5.1.1.9.1~~5.1.1.11.1 `geo.distance`

The `geo.distance` function has the following signatures:

```
Edm.Double geo.distance(Edm.GeographyPoint,Edm.GeographyPoint)
Edm.Double geo.distance(Edm.GeometryPoint,Edm.GeometryPoint)
```

The `geo.distance` function returns the shortest distance between the two points in the coordinate reference system signified by the two points' SRIDs.

~~5.1.1.9.2~~5.1.1.11.2 `geo.intersects`

The `geo.intersects` function has the following signatures:

```
Edm.Boolean geo.intersects(Edm.GeographyPoint,Edm.GeographyPolygon)
Edm.Boolean geo.intersects(Edm.GeometryPoint,Edm.GeometryPolygon)
```

The `geo.intersects` function returns true if the specified point lies within the interior or on the boundary of the specified polygon, otherwise it returns false.

~~5.1.1.9.3~~5.1.1.11.3 `geo.length`

The `geo.length` function has the following signatures:

```
Edm.Double geo.length(Edm.GeographyLineString)
Edm.Double geo.length(Edm.GeometryLineString)
```

The `geo.length` function returns the total length of its line string parameter in the coordinate reference system signified by its SRID.

5.1.1.12 Lambda Operators

OData defines two operators that evaluate a Boolean expression on a collection. Both must be prepended with a navigation path that identifies a collection.

4.01 Services MUST support case-insensitive lambda operator names. Clients that want to work with 4.0 services MUST use lower case lambda operator names.

The argument of a lambda operator is a case-sensitive lambda variable name followed by a colon (`:`) and a Boolean expression that uses the lambda variable name to refer to properties of members of the collection identified by the resource path.

If the name chosen for the lambda variable matches a property name of the current resource referenced by the resource path, the lambda variable takes precedence. Clients can prefix properties of the current resource referenced by the resource path with `$it`.

~~5.1.1.9.4~~5.1.1.12.1 `any`

The `any` operator applies a Boolean expression to each member of a collection and returns true if and only if the expression is true for any member of the collection, otherwise it returns false. This implies that the `any` operator always returns false for an empty collection.

The `any` operator can be used without an argument expression. This short form returns ~~true~~ false if and only if the collection is ~~not~~ empty.

Example 92: all `Orders` that have any `Items` with a `Quantity` greater than `100`

```
http://host/service/Orders?$filter=Items/any(d:d/Quantity gt 100)
```

~~5.1.1.9.5~~5.1.1.12.2 `all`

The `all` operator applies a Boolean expression to each member of a collection and returns true if the expression is true for all members of the collection, otherwise it returns false. This implies that the `all` operator always returns true for an empty collection.

The `all` operator cannot be used without an argument expression.

Example 93: all `Orders` that have only `Items` with a `Quantity` greater than `100`

```
http://host/service/Orders?$filter=Items/all(d:d/Quantity gt 100)
```

~~5.1.1.10~~5.1.1.13 Literals

~~5.1.1.10.1~~5.1.1.13.1 Primitive Literals

Primitive literals can appear in the resource path as key property values, and in the query part, for example, as operands in `$filter` expressions. They are represented according to the `primitiveLiteral` rule in **[OData-ABNF]**.

Example 94: expressions using primitive literals

```
NullValue eq null
TrueValue eq true
FalseValue eq false
Custom.Base64UrlDecode(binary'T0RhdGE') eq 'OData'
IntegerValue lt -128
DoubleValue ge 0.31415926535897931e1
SingleValue eq INF
DecimalValue eq 34.95
StringValue eq 'Say Hello,then go'
DateValue eq 2012-12-03
DateTimeOffsetValue eq 2012-12-03T07:16:23Z
DurationValue eq duration'P12DT23H59M59.999999999999S'
DurationValue eq 'P12DT23H59M59.999999999999S'
TimeOfDayValue eq 07:59:59.999
GuidValue eq 01234567-89ab-cdef-0123-456789abcdef
Int64Value eq 0
ColorEnumValue eq Sales.Pattern'Yellow',
ColorEnumValue eq 'Yellow',
geo.distance(Location,geography'SRID=0;Point(142.1 64.1)')
```

Duration literals in OData 4.0 required prefixing with “duration”. Enumeration literals in OData 4.0 required prefixing with the qualified type name of the enumeration.

In OData 4.01, services MUST support duration and enumeration literals with or without the type prefix. OData clients that want to operate across OData 4.0 and OData 4.01 services should always include the prefix for duration and enumeration types.

~~5.1.1.10.2~~5.1.1.13.2 Complex and Collection Literals

Complex literals and collection literals in URLs are represented as JSON objects and arrays according to the `arrayOrObject` rule in **[OData-ABNF]**. Such literals MUST NOT appear in the path portion of the URL but can be passed to bound functions and function imports in path segments by using parameter aliases.

Note that the special characters `{`, `}`, `[`, `]`, and `"` MUST be percent-encoded in URLs although some browsers will accept and pass them on unencoded.

Example 95: collection of string literals

```
http://host/service/ProductsByColors(colors=@c)?@c=["red","green"]
```

~~5.1.1.10.3~~5.1.1.13.3 `null`

The `null` literal can be used to compare a value to null, or to pass a null value to a function.

~~5.1.1.10.4~~5.1.1.13.4 `$it`

The `$it` literal can be used in expressions to refer to the current instance of the resource ~~collection~~ identified by the resource path. For a collection-valued resource the current instance is the instance currently being evaluated by the system query option ~~is evaluated on~~. For a single-valued resource it is the resource itself.

It ~~can be used to~~allows comparing~~e~~ properties of related entities to properties of the current instance in expressions within lambda operators, for example in `$filter` and `$orderby` expressions on collections of primitive types, or in `$filter` expressions nested within `$expand`.

Note: property names and property paths in `$filter` expressions nested within `$expand` are evaluated in the context of the declared type of the expanded navigation property, so property names and property paths for the current instance of the collection identified by the resource path MUST be prefixed with `$it/`.

The `$it` literal~~It~~ can also be used as a path prefix to invoke a bound function overload on the current instance within an expression. Function names without a path prefix refer to an unbound function overload.

Example 96: email addresses ending with `.com` assuming `EmailAddresses` is a collection of strings

```
http://host/service/Customers(1)/EmailAddresses?$filter=endswith($it,'.com')
```

Example 97: customers along with their orders that shipped to the same city as the customer's address. The nested filter expression is evaluated in the context of Orders; `$it` allows referring to values in the outer context of Customers.

```
http://host/service/Customers?
            $expand=Orders($filter=$it/Address/City eq ShipTo/City)
```

Example 98: products with at least 10 positive reviews. `Model.PositiveReviews` is a function bound to `Model.Product` returning a collection of reviews.

```
http://host/service/Products?$filter=$it/Model.PositiveReviews()/$count ge 10
```

~~5.1.1.10.5~~5.1.1.13.5 `$root`

The `$root` literal can be used in expressions to refer to resources of the same service. It can be used as a single-valued expression or within complex or collection literals.

Example 99: all employees with the same last name as employee `A1235`

```
http://host/service/Employees?
                         $filter=LastName eq $root/Employees('A1245')/LastName
```

Example 100: products ordered by a set of customers, where the set of customers is passed as a JSON array containing the resource paths from `$root` to each customer.

```
http://host/service/ProductsOrderedBy(Customers=@c)?
                        @c=[$root/Customers('ALFKI'),$root/Customers('BLAUS')]
```

~~5.1.1.11~~5.1.1.14 Path Expressions

Properties and navigation properties of the entity type of the set of resources that are addressed by the request URL can be used as operands or function parameters, as shown in the preceding examples.

Properties of complex properties can be used via the same syntax as in resource paths, i.e. by specifying the name of a complex property, followed by a forward slash (`/`) and the name of a property of the complex property, and so on,

Properties and navigation properties of entities related with a target cardinality 0..1 or 1 can be used by specifying the navigation property, followed by a forward slash (`/`) and the name of a property of the related entity, and so on.

If a complex property is `null`, or no entity is related (in case of target cardinality 0..1), its value, and the values of its components, are treated as `null`.

Example 101: similar behavior whether `HeadquarterAddress` is a nullable complex type or a nullable navigation property

```
Companies(1)/HeadquarterAddress/Street
```

To access properties of derived types, the property name MUST be prefixed with the qualified name of the derived type on which the property is defined, followed by a forward slash (`/`), see addressing derived types. If the current instance is not of the specified derived type, the path expression returns `null`.

If the property or navigation property is not defined for the type of the resource and that type supports dynamic properties or navigation properties, then the property or navigation property is treated as null for all instances on which it has no value.

If the property or navigation property is not defined for the type of the resource and that type does not support dynamic properties or navigation properties, then the request may be considered malformed.

~~5.1.1.12~~5.1.1.15 Annotation Values in Expressions

Services MAY support the use of annotation values as operands or function parameters.

Annotation values are referenced by the annotation name which consists of an at sign (`@`) followed by the qualified term name, optionally followed by a hash (`#`) and an annotation qualifier~~,~~. The annotation name can be prefixed with a path expression (see section 5.1.1.14) leading to the annotated resource or property.

If an annotation is not applied to the resource or property, then its value, and the values of its components, are treated as `null`.

Example 102: Return Products that have prices in Euro

```
http://host/service/Products?
    $filter=Price/@Measures.Currency eq 'EUR'
```

Example 90: Return Employees that have any error messages in the Core.Message annotation

```
http://host/service/Employees?
    $filter=@Core.Messages/any(m:m/severity eq 'error')
```

Services MAY additionally support the use of the unqualified term name by defining one or more default namespaces through the `DefaultNamespace` annotation term defined in **[OData-VocCore]**. For more information on default namespaces, see Default Namespaces in **[OData-Protocol]**. This short notation however uses the same name pattern as parameter aliases. If a query option is specified as a parameter alias, then any occurrence of the parameter alias name in an expression MUST evaluate to the parameter alias value and MUST NOT evaluate to the annotation value of an identical unqualified term name.

~~5.1.1.13~~5.1.1.16 Operator Precedence

OData services MUST use the following operator precedence for supported operators when evaluating `$filter` and `$orderby` expressions. Operators are listed by category in order of precedence from highest to lowest. Operators in the same category have equal precedence:

Group	Operator	Description	ABNF Expression
Grouping	`( )`	Precedence grouping	`parenExpr` `boolParenExpr`
Primary	`/`	Navigation	`firstMemberExpr` `memberExpr`
	`has`	Enumeration Flags	`hasExpr`
	`in`	Is a member of	`inExpr`

	`xxx( )`	Method Call	`methodCallExpr`
			`boolMethodCallExpr`
			`functionExpr`

Unary	`-`	Negation	`negateExpr`
	`not`	Logical Negation	`notExpr`
	`cast( )`	Type Casting	`castExpr`
Multiplicative	`mul`	Multiplication	`mulExpr`
	`div`	Division	`divExpr`
	`divby`	Decimal Division	`divbyExpr`
	`mod`	Modulo	`modExpr`
Additive	`add`	Addition	`addExpr`
	`sub`	Subtraction	`subExpr`
Relational	`gt`	Greater Than	`gtExpr`
	`ge`	Greater than or Equal	`geExpr`
	`lt`	Less Than	`ltExpr`
	`le`	Less than or Equal	`leExpr`
	`isof`	Type Testing	`isofExpr`
Equality	`eq`	Equal	`eqExpr`
	`ne`	Not Equal	`neExpr`
Conditional AND	`and`	Logical And	`andExpr`
Conditional OR	`or`	Logical Or	`orExpr`

~~5.1.1.14~~5.1.1.17 Numeric Promotion

Services SHOULD NOT require explicit cast operations between numeric types used in comparison expressions. Wherever possible, such comparisons should be performed using underlying types of sufficient size.

Services MAY support numeric promotion for arithmetic operations or when comparing two operands of comparable types by applying the following rules, in order:

- If either operand is `Edm.Double`, the other operand is converted to type `Edm.Double`.
- Otherwise, if either operand `is Edm.Single`, the other operand is converted to type `Edm.Single`.
- Otherwise, if either operand is of type `Edm.Decimal`, the other operand is converted to `Edm.Decimal`.
- Otherwise, if either operand is `Edm.Int64`, the other operand is converted to type `Edm.Int64`.
- Otherwise, if either operand is `Edm.Int32`, the other operand is converted to type `Edm.Int32`.
- Otherwise, if either operand is `Edm.Int16`, the other operand is converted to type `Edm.Int16`.

Each of these promotions uses the same semantics as a `castExpression` to promote an operand to the target type.

OData does not define an implicit conversion between string and numeric types.

5.1.2 System Query Option `$expand`

The `$expand` system query option specifies the related resources or media streams to be included in line with retrieved resources.

What follows is a (non-normative) snippet from **[OData-ABNF]** that describes the syntax of `$expand`:

```
expand            = '$expand' EQ expandItem *( COMMA expandItem )

expandItem        = STAR [ ref / OPEN levels CLOSE ]
                  / "$value"
                  / expandPath
                    [ ref   [ OPEN expandRefOption
                             *( SEMI expandRefOption   ) CLOSE ]
                    / count [ OPEN expandCountOption
                             *( SEMI expandCountOption ) CLOSE ]
                    /         OPEN expandOption
                             *( SEMI expandOption      ) CLOSE
                    ]

expandPath        = [ ( qualifiedEntityTypeName
                      / qualifiedComplexTypeName
                      ) "/" ]
                    *( ( complexProperty / complexColProperty ) "/"
                       [ qualifiedComplexTypeName "/" ] )
                    ( STAR
                    / streamProperty
                    / navigationProperty [ "/" qualifiedEntityTypeName ]
                    )

expandCountOption = filter
                  / search

expandRefOption   = expandCountOption
                  / orderby
                  / skip
                  / top
                  / inlinecount
```

```
expandOption       = expandRefOption
                   / select
                   / expand
                   / levels
```

Each `expandItem` is evaluated relative to the entity containing the navigation or stream property being expanded.

A type cast using the `qualifiedEntityTypeName` to a type containing the property is required in order to expand a navigation or stream property defined on a derived type.

An arbitrary number of single- or collection-valued complex properties, optionally followed by a type cast, allow drilling into complex properties.

If the type does not support dynamic properties, then the `navigationProperty` or `streamProperty` segment MUST identify a navigation or stream property defined on the entity type of the request, the derived entity type specified in the type cast, or the last complex type identified by the complex property path. Otherwise, if the type does support dynamic navigation properties and the `navigationProperty` or `streamProperty` segment does not identify a declared navigation or stream property, then the expanded property appears only for those instances on which it has a value.

Example 103: expand a navigation property of an entity type

```
http://host/service/Products?$expand=Category
```

Example 104: expand a navigation property of a complex type

```
http://host/service/Customers?$expand=Addresses/Country
```

A property MUST NOT appear in more than one `expandItem`.

Query options can be applied to an expanded navigation property by appending a semicolon-separated list of query options, enclosed in parentheses, to the navigation property name. Allowed system query options are `$filter`, `$select`, `$orderby`, `$skip`, `$top`, `$count`, `$search`, and `$expand`.

Example 105: all categories and for each category all related products with a discontinued date equal to `null`

```
http://host/service/Categories?
                                 $expand=Products($filter=DiscontinuedDate eq null)
```

The `$count` segment can be appended to a navigation property name or type-cast segment following a navigation property name to return just the count of the related entities. The `$filter` and `$search` system query options can be used to limit the number or related entities included in the count.

Example 106: all categories and for each category the number of all related products

```
http://host/service/Categories?$expand=Products/$count
```

Example 107: all categories and for each category the number of all related blue products

```
http://host/service/Categories?$expand=Products/$count($search=blue)
```

To retrieve entity references instead of the related entities, append `/$ref` to the navigation property name or type-cast segment following a navigation property name.

Example 108: all categories and for each category the references of all related products

```
http://host/service/Categories?$expand=Products/$ref
```

Example 109: all categories and for each category the references of all related products of the derived type `Sales.PremierProduct`

```
http://host/service/Categories?$expand=Products/Sales.PremierProduct/$ref
```

Example 110: all categories and for each category the references of all related premier products with a current promotion equal to `null`

```
http://host/service/Categories?
   $expand=Products/Sales.PremierProduct/$ref($filter=CurrentPromotion eq null)
```

Cyclic navigation properties (whose target type is identical or can be cast to its source type) can be recursively expanded using the special `$levels` option. The value of the `$levels` option is either a positive integer to specify the number of levels to expand, or the literal string `max` to specify the maximum expansion level supported by that service. A `$levels` option with a value of 1 specifies a single expand with no recursion.

Example 111: all employees with their manager, manager's manager, and manager's manager's manager

```
http://host/service/Employees?$expand=ReportsTo($levels=3)
```

It is also possible to expand all declared and dynamic navigation properties using a star (`*`). To retrieve references to all related entities use `*/$ref`, and to expand all related entities with a certain distance use the star operator with the `$levels` option. The star operator can be combined with explicitly named navigation properties, which take precedence over the star operator.

The star operator does not implicitly include stream properties.

Example 112: expand `Supplier` and include references for all other related entities

```
http://host/service/Categories?$expand=*/$ref,Supplier
```

Example 113: expand all related entities and their related entities

```
http://host/service/Categories?$expand=*($levels=2)
```

Specifying a stream property includes the media stream inline according to the specified format.

Example 114: include Employee's `Photo` stream property along with other properties of the customer

```
http://host/service/Employees?$expand=Photo
```

Specifying `$value` for a media entity includes the media entity's stream value inline according to the specified format.

Example 115: Include the `Product's` media stream along with other properties of the product

```
http://host/service/Products?$expand=$value
```

5.1.3 System Query Option `$select`

The `$select` system query option allows clients to requests a specific set of properties for each entity or complex type.

The `$select` query option is often used in conjunction with the `$expand` system query option, to define the extent of the resource graph to return (`$expand`) and then specify a subset of properties for each resource in the graph (`$select`). Expanded navigation properties MUST be returned, even if they are not specified as a `selectItem`.

What follows is a (non-normative) snippet from **[OData-ABNF]** showing the syntax of `$select`:

```
select         = '$select' EQ selectItem *( COMMA selectItem )

selectItem     = STAR
               / allOperationsInSchema
               / [ ( qualifiedEntityTypeName

                   / qualifiedComplexTypeName

                   ) "/" ]
                 ( selectProperty
                 / qualifiedActionName
                 / qualifiedFunctionName
                 )

selectProperty = primitiveProperty
               / primitiveColProperty
               / navigationProperty
               / selectPath [ "/" selectProperty ]

selectPath     = ( complexProperty / complexColProperty )
                 [ "/" qualifiedComplexTypeName ]
```

The `$select` system query option is interpreted relative to the entity type or complex type of the resources identified by the resource path section of the URL. Each `selectItem` in the `$select` clause indicates that the response MUST include the declared or dynamic properties, actions and functions identified by that `selectItem`. The simplest form of a `selectItem` explicitly requests a property defined on the entity type of the resources identified by the resource path section of the URL.

Example 116: rating and release date of all products

```
http://host/service/Products?$select=Rating,ReleaseDate
```

It is also possible to request all declared and dynamic structural properties using a star (`*`).

Example 117: all structural properties of all products

```
http://host/service/Products?$select=*
```

If the `selectItem` is not defined for the type of the resource, and that type supports dynamic properties, then the property is treated as null for all instances on which it is not defined.

If the `selectItem` is not defined for the type of the resource, and that type does not support dynamic properties, then the request is considered malformed.

If the `selectItem` is a navigation property then the corresponding navigation link is represented in the response. If the navigation property also appears in an `$expand` query option then it is additionally represented as inline content. This inline content can itself be restricted with a nested `$select` query option, see section 5.1.2.

Example 118: name and description of all products, plus name of expanded category

```
http://host/service/Products?
                           $select=Name,Description&$expand=Category($select=Name)
```

The `selectItem` MUST be prefixed with a `qualifiedEntityTypeName` or `qualifiedComplexTypeName` in order to select a property defined on a type derived from the type of the resource segment.

A `selectItem` that is a complex type or collection of complex type can be followed by a forward slash, an optional type cast segment, and the name of a property of the complex type (and so on for nested complex types).

Example 119: the `AccountRepresentative` property of any supplier that is of the derived type `Namespace.PreferredSupplier`, together with the `Street` property of the complex property `Address`, and the `Location` property of the derived complex type `Namespace.AddressWithLocation`

```
http://host/service/Suppliers?
      $select=Namespace.PreferredSupplier/AccountRepresentative,
               Address/Street,
               Address/Namespace.AddressWithLocation/Location
```

Any structural property, non-expanded navigation property, or operation not requested as a `selectItem` (explicitly or via a star) SHOULD be omitted from the response.

If any `selectItem` (including a star) is specified, actions and functions SHOULD be omitted unless explicitly requested using a `qualifiedActionName`, a `qualifiedFunctionName` or the `allOperationsInSchema`.

If an action or function is requested as a `selectItem`, either explicitly by using a `qualifiedActionName` or `qualifiedFunctionName` clause, or implicitly by using `allOperationsInSchema`, then the service includes information about how to invoke that operation for each entity identified by the last path segment in the request URL for which the operation can be bound.

If an action or function is requested in a `selectItem` using a `qualifiedActionName` or a `qualifiedFunctionName` and that operation cannot be bound to the entities requested, the service MUST ignore the `selectItem`.

Example 120: the `ID` property, the `ActionName` action defined in `Model` and all actions and functions defined in the `Model2` for each product if those actions and functions can be bound to that product

```
http://host/service/Products?$select=ID,Model.ActionName,Model2.*
```

When multiple `selectItems` exist in a select clause, then the total set of properties, open properties, navigation properties, actions and functions to be returned is equal to the union of the set of those identified by each `selectItem`.

If a `selectItem` is a path expression requesting a component of a complex property and the complex property is `null` on an instance, then the component is treated as `null` as well.

5.1.4 System Query Option `$orderby`

The `$orderby` system query option allows clients to request resources in a particular order.

The semantics of `$orderby` are covered in the **[OData-Protocol]** document.

The **[OData-ABNF]** `orderby` syntax rule defines the formal grammar of the `$orderby` query option.

5.1.5 System Query Options `$top` and `$skip`

The `$top` system query option requests the number of items in the queried collection to be included in the result. The `$skip` query option requests the number of items in the queried collection that are to be skipped and not included in the result. A client can request a particular page of items by combining `$top` and `$skip`.

The semantics of `$top` and `$skip` are covered in the **[OData-Protocol]** document. The **[OData-ABNF]** `top` and `skip` syntax rules define the formal grammar of the `$top` and `$skip` query options respectively.

5.1.6 System Query Option `$count`

The `$count` system query option allows clients to request a count of the matching resources included with the resources in the response. The `$count` query option has a Boolean value of `true` or `false`.

The semantics of `$count` is covered in the **[OData-Protocol]** document.

5.1.7 System Query Option `$search`

The `$search` system query option allows clients to request items within a collection matching a free-text search expression.

The `$search` query option can be applied to a URL representing a collection of entity, complex, or primitive typed instances, to return all matching items within the collection. Applying the `$search` query option to the `$all` resource requests all matching entities in the service.

If both `$search` and `$filter` are applied to the same request, the results include only those items that match both criteria.

The **[OData-ABNF]** `search` syntax rule defines the formal grammar of the `$search` query option.

Example 121: all products that are blue or green. It is up to the service to decide what makes a product blue or green.

```
http://host/service/Products?$search=blue OR green
```

5.1.7.1 Search Expressions

Search expressions are used within the `$search` system query option to request entities matching the specified expression.

Terms can be any single word to be matched within the expression.

Terms enclosed in double-quotes comprise a *phrase*.

Each individual term or phrase comprises a Boolean expression that returns true if the term or phrase is matched, otherwise false. The semantics of what is considered a match is dependent upon the service.

Expressions enclosed in parenthesis comprise a *group expression*.

The search expression can contain any number of terms, phrases, or group expressions, along with the case-sensitive keywords `NOT`, `AND`, and `OR`, evaluated in that order.

Expressions prefaced with `NOT` evaluate to true if the expression is not matched, otherwise false.

Two expressions not enclosed in quotes and separated by a space are equivalent to the same two expressions separated by the `AND` keyword. Such expressions evaluate to `true` if both of the expressions evaluate to true, otherwise false.

Expressions separated by an `OR` evaluate to true if either of the expressions evaluate to true, otherwise false.

The **[OData-ABNF]** `searchExpr` syntax rule defines the formal grammar of the search expression.

5.1.8 System Query Option `$format`

The `$format` system query option allows clients to request a response in a particular format and is useful for clients without access to request headers for standard content-type negotiation. Where present `$format` takes precedence over standard content-type negotiation.

The semantics of `$format` is covered in the **[OData-Protocol]** document.

The **[OData-ABNF]** `format` syntax rule defines the formal grammar of the `$format` query option.

5.1.9 System Query Option `$compute`

The `$compute` system query option allows clients to define computed properties that can be used in a `$select` or within a `$filter` or `$orderby` expression.

Following is a (non-normative) snippet from **[OData-ABNF]** showing the syntax of `$compute`:

```
compute          = '$compute' EQ computeItem *( COMMA computeItem )
computeItem      = commonExpr RWS 'as' RWS computedProperty
computedProperty = odataIdentifier
```

The `$compute` system query option is interpreted relative to the entity type or complex type of the resources identified by the resource path section of the URL.

Computed properties SHOULD be included as dynamic properties in the result and MUST be included if `$select` is specified with the computed property name, or star (`*`).

Example 122: compute total price for order items

```
http://host/service/Orders(10)/Items?$select=Product/Description,Total
       &$filter=Total gt 100&$orderby=Total
       &$compute=Product/Price mul Quantity as Total
```

5.1.10 System Query Option `$index`

The `$index` system query option allows clients to do a positional insert into a collection annotated with using the `Core.PositionalInsert` term (see **[OData-VocCore]**). The value of the `$index` system query option is the zero-based ordinal position where the item is to be inserted. The ordinal of items within the collection greater than or equal to the inserted position are increased by one. A negative ordinal indexes from the end of the collection, with -1 representing an insert at the end of the collection.

The **[OData-ABNF]** `index` syntax rule defines the formal grammar of the `$index` query option.

5.1.11 System Query Option `$schemaversion`

The `$schemaversion` system query option allows clients to specify the version of the schema against which the request is made. The semantics of `$schemaversion` is covered in the **[OData-Protocol]** document.

The **[OData-ABNF]** `schemaversion` syntax rule defines the formal grammar of the `$schemaversion` query option

5.2 Custom Query Options

Custom query options provide an extensible mechanism for service-specific information to be placed in a URL query string. A custom query option is any query option of the form shown by the rule `customQueryOption` in **[OData-ABNF]**.

Custom query options MUST NOT begin with a `$` or `@` character.

Example 123: service-specific custom query option `debug-mode`

```
http://host/service/Products?debug-mode=true
```

5.3 Parameter Aliases

Parameter aliases can be used in place of literal values in entity keys, function parameters, or within a `$filter` or `$orderby` expression.

Parameter aliases MUST start with an `@` character, see rule `parameterAlias` in **[OData-ABNF]**.

The semantics of parameter aliases are covered in **[OData-Protocol]**.

The **[OData-ABNF]** rule `aliasAndValue` defines the formal grammar for passing parameter alias value~~e~~s as query options.

~~Parameter aliases can be used within `$filter` or `$orderby` in place of expressions that evaluate to a primitive value, a complex value, or a collection of primitive or complex values. Parameter names start with the at sign (`@`) and can be used in more than one place in the expression. The value for the parameter alias is supplied in a query option with the same name as the parameter.~~

Example 124:

```
http://host/service/Movies?$filter=contains(@word,Title)&@word='Black'
```

Example 125:

```
http://host/service/Movies?$filter=Title eq @title&@title='Wizard of Oz'
```

Example 126: JSON array of strings as parameter alias value – note that `[`, `]`, and `"` need to be percent-encoded in real URLs, the clear-text representation used here is just for readability

```
http://host/service/Products/Model.WithIngredients(Ingredients=@i)?@i=["Carrot
s","Ginger","Oranges"]
```

6 Conformance

The conformance requirements for OData clients and services are described in **[OData-Protocol]**.

Appendix A. Acknowledgments

The contributions of the OASIS OData Technical Committee members, enumerated in **[OData-Protocol]**, are gratefully acknowledged.

Appendix B. Revision History

Revision	Date	Editor	Changes Made
Working Draft 01	2016-06-22~~2012-09-27~~	Michael Pizzo Ralf Handl	Transferred content from OData 4.0 Part 2 – URL Conventions Errata 3 ~~Integrated initial 4.01 features~~
Committee Specification Draft 01	2016-12-08	Michael Pizzo Ralf Handl	Integrated 4.01 features • Alternate keys • Key-as-segment convention • Operators in and `divby` • `$expand` with stream properties • `$compute` system query option • `$index` system query option
Committee Specification Draft 02	2017-06-08	Michael Pizzo Ralf Handl	More 4.01 features • Set-based operations • `$expand`, `$select`, `$filter` with `POST` and `PATCH` • Instance annotations in query options • `$schemaversion` system query option