OData Extension for Data Aggregation Version 4.0

Working Draft 05

20 December 2017

Technical Committee:

[OASIS Open Data Protocol (OData) TC](http://www.oasis-open.org/committees/odata/)

Chairs:

Ralf Handl (ralf.handl@sap.com), [SAP SE](http://www.sap.com/)

Ram Jeyaraman (Ram.Jeyaraman@microsoft.com), [Microsoft](http://www.microsoft.com/)

Editor:

Ralf Handl (ralf.handl@sap.com), [SAP SE](http://www.sap.com/)

Hubert Heijkers (hubert.heijkers@nl.ibm.com), [IBM](http://www.ibm.com/)

Gerald Krause (gerald.krause@sap.com), [SAP SE](http://www.sap.com/)

Michael Pizzo (mikep@microsoft.com), [Microsoft](http://www.microsoft.com/)

Martin Zurmuehl (martin.zurmuehl@sap.com), [SAP SE](http://www.sap.com/)

Additional artifacts:

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

* OData Extension for Data Aggregation Version 4.0 (this document)
* [OData Aggregation ABNF Construction Rules](https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/ABNF/odata-aggregation-abnf.txt) Version 4.0
* [OData Aggregation ABNF Test Cases](https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/ABNF/odata-aggregation-testcases.xml)
* [OData Aggregation Vocabulary](https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/vocabularies/Org.OData.Aggregation.V1.xml)

Related work:

This specification is related to:

* OData Version 4.0 Part 1: Protocol
* OData Version 4.0 Part 2: URL Conventions
* OData Version 4.0 Part 3: CSDL
* [OData ABNF Construction Rules](https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/ABNF/odata-abnf-construction-rules.txt) Version 4.0
* [OData ABNF Test Cases](https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/ABNF/odata-abnf-testcases.xml)
* [OData Core Vocabulary](https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/vocabularies/Org.OData.Core.V1.xml)
* [OData Measures Vocabulary](https://tools.oasis-open.org/version-control/browse/wsvn/odata/trunk/spec/vocabularies/Org.OData.Measures.V1.xml)
* OData JSON Format Version 4.0

This specification replaces or supersedes:

* None

Declared XML namespaces:

* None

Abstract:

This specification adds basic grouping and aggregation functionality (e.g. sum, min, and max) to the Open Data Protocol (OData) without changing any of the base principles of OData.

Status:

This [Working Draft](http://www.oasis-open.org/committees/process.php#dWorkingDraft) (WD) has been produced by one or more TC Members; it has not yet been voted on by the TC or [approved](http://www.oasis-open.org/committees/process.php#committeeDraft) as a Committee Draft (Committee Specification Draft or a Committee Note Draft). The OASIS document [Approval Process](http://www.oasis-open.org/committees/process.php#standApprovProcess) 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.

Copyright © OASIS Open 2016. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](http://www.oasis-open.org/policies-guidelines/ipr) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

[1 Introduction 5](#_Toc492655023)

[1.1 Terminology 5](#_Toc492655024)

[1.2 Normative References 5](#_Toc492655025)

[1.3 Non-Normative References 5](#_Toc492655026)

[1.4 Typographical Conventions 5](#_Toc492655027)

[2 Overview 6](#_Toc492655028)

[2.1 Definitions 6](#_Toc492655029)

[2.2 Example Data Model 7](#_Toc492655030)

[2.3 Example Data 8](#_Toc492655031)

[2.4 Example Use Cases 8](#_Toc492655032)

[3 System Query Option $apply 10](#_Toc492655033)

[3.1 Transformation aggregate 11](#_Toc492655034)

[3.1.1 Keyword as 11](#_Toc492655035)

[3.1.2 Keyword with 12](#_Toc492655036)

[3.1.3 Aggregation Methods 12](#_Toc492655037)

[3.1.3.1 Standard Aggregation Method sum 12](#_Toc492655038)

[3.1.3.2 Standard Aggregation Method min 13](#_Toc492655039)

[3.1.3.3 Standard Aggregation Method max 13](#_Toc492655040)

[3.1.3.4 Standard Aggregation Method average 13](#_Toc492655041)

[3.1.3.5 Standard Aggregation Method countdistinct 14](#_Toc492655042)

[3.1.3.6 Custom Aggregation Methods 14](#_Toc492655043)

[3.1.4 Keyword from 14](#_Toc492655044)

[3.1.5 Virtual Property $count 15](#_Toc492655045)

[3.2 Transformation topcount 16](#_Toc492655046)

[3.3 Transformation topsum 16](#_Toc492655047)

[3.4 Transformation toppercent 17](#_Toc492655048)

[3.5 Transformation bottomcount 17](#_Toc492655049)

[3.6 Transformation bottomsum 18](#_Toc492655050)

[3.7 Transformation bottompercent 18](#_Toc492655051)

[3.8 Transformation identity 19](#_Toc492655052)

[3.9 Transformation concat 19](#_Toc492655053)

[3.10 Transformation groupby 20](#_Toc492655054)

[3.10.1 Simple Grouping 20](#_Toc492655055)

[3.10.2 Grouping with rollup and $all 21](#_Toc492655056)

[3.11 Transformation filter 23](#_Toc492655057)

[3.12 Transformation expand 23](#_Toc492655058)

[3.13 Transformation search 24](#_Toc492655059)

[3.14 Transformation compute 24](#_Toc492655060)

[3.15 Filter Function isdefined 26](#_Toc492655061)

[3.16 Evaluating $apply 27](#_Toc492655062)

[3.17 Evaluating $apply as an Expand Option 27](#_Toc492655063)

[3.18 ABNF for Extended URL Conventions 27](#_Toc492655064)

[4 Representation of Aggregated Instances 28](#_Toc492655065)

[5 Cross-Joins and Aggregation 29](#_Toc492655066)

[6 Vocabulary for Data Aggregation 30](#_Toc492655067)

[6.1 Aggregation Capabilities 30](#_Toc492655068)

[6.2 Property Annotations 30](#_Toc492655069)

[6.2.1 Groupable Properties 30](#_Toc492655070)

[6.2.2 Aggregatable Properties 30](#_Toc492655071)

[6.2.3 Custom Aggregates 31](#_Toc492655072)

[6.2.4 Context-Defining Properties 32](#_Toc492655073)

[6.2.5 Annotation Example 32](#_Toc492655074)

[6.3 Hierarchies 33](#_Toc492655075)

[6.3.1 Leveled Hierarchy 33](#_Toc492655076)

[6.3.2 Recursive Hierarchy 33](#_Toc492655077)

[6.3.2.1 Hierarchy Filter Functions 33](#_Toc492655078)

[6.3.3 Hierarchy Examples 34](#_Toc492655079)

[6.4 Actions and Functions on Aggregated Entities 36](#_Toc492655080)

[7 Examples 38](#_Toc492655081)

[7.1 Requesting Distinct Values 38](#_Toc492655082)

[7.2 Standard Aggregation Methods 39](#_Toc492655083)

[7.3 Requesting Custom Aggregates 42](#_Toc492655084)

[7.4 Aliasing 43](#_Toc492655085)

[7.5 Combining Transformations per Group 44](#_Toc492655086)

[7.6 Model Functions as Set Transformations 45](#_Toc492655087)

[7.7 Controlling Aggregation per Rollup Level 46](#_Toc492655088)

[7.8 Transformation Sequences 47](#_Toc492655089)

[8 Conformance 50](#_Toc492655090)

[Appendix A. Acknowledgments 51](#_Toc492655091)

[Appendix B. Revision History 52](#_Toc492655092)

# [Introduction](#sec_Introduction)

This specification adds the notion of aggregation to the Open Data Protocol (OData) without changing any of the base principles of OData. It defines semantics and a representation for aggregation of data, especially:

* Semantics and operations for querying aggregated data,
* Results format for queries containing aggregated data,
* Vocabulary terms to annotate what can be aggregated, and how.

## [Terminology](#sec_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]**](#rfc2119)**.**

## [Normative References](#sec_NormativeReferences)

[OData-ABNF] *OData ABNF Construction Rules Version 4.0*.
See the link in "Related work" section on cover page.

[OData-Agg-ABNF] *OData Aggregation ABNF Construction Rules Version 4.0*.
See link in "Additional artifacts" section on cover page.

[OData-CSDL] *OData Version 4.0 Part 3: CSDL*.
See link in "Related work" section on cover page.

**[****OData-JSON]** *OData JSON Format Version 4.0.*
See link in "Related work" section on cover page.

[OData-Protocol] *OData Version 4.0 Part 1: Protocol*.
See link in "Related work" section on cover page.

[OData-URL] *OData Version 4.0 Part 2: URL Conventions*.
See link in "Related work" section on cover page.

**[****OData-VocAggr]** *OData Aggregation Vocabulary*.
See link in "Additional artifacts" section on cover page.

**[****OData-VocMeas]** *OData Measures 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. <http://www.ietf.org/rfc/rfc2119.txt>.

## [Non-Normative References](#sec_NonNormativeReferences)

[TSQL ROLLUP] <http://msdn.microsoft.com/en-us/library/bb522495.aspx>

## [Typographical Conventions](#sec_TypographicalConventions)

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.

# [Overview](#sec_Overview)

Open Data (OData) services expose a data model that describes the schema of the service in terms of the Entity Data Model (EDM, see [**[OData-CSDL]**](#ODataCSDL))and then allows for querying data in terms of this model. The responses returned by an OData service are based on that data model and retain the relationships between the entities in the model.

Extending the OData query features with simple aggregation capabilities avoids cluttering OData services with an exponential number of explicitly modeled “aggregation level entities” or else restricting the consumer to a small subset of predefined aggregations.

Adding the notion of aggregation to OData without changing any of the base principles in OData has two aspects:

1. Means for the consumer to query aggregated data on top of any given data model (for sufficiently capable data providers)
2. Means for the provider to annotate what data can be aggregated, and in which way, allowing consumers to avoid asking questions that the provider cannot answer.

Implementing any of these two aspects is valuable in itself independent of the other, and implementing both provides additional value for consumers. The descriptions provided by the provider help a consumer understand more of the data structure looking at the service's exposed data model. The query extensions allow the consumers to express explicitly the desired aggregation behavior for a particular query. They also allow consumers to formulate queries that refer to the annotations as shorthand.

## [Definitions](#sec_Definitions)

This specification defines the following terms:

* *[Aggregatable Property](#sec_AggregatableProperties)* – a property for which the values can be aggregated using an aggregation method
* [*Aggregation Method*](#sec_AggregationMethods) *–* a method that can be used to aggregate an aggregatable property or expression
* *Standard Aggregation Method –* one of the standard aggregation methods: [sum](#sec_StandardAggregationMethodsum), [min](#sec_StandardAggregationMethodmin), [max](#sec_StandardAggregationMethodmax), [average](#sec_StandardAggregationMethodaverage), and [countdistinct](#sec_StandardAggregationMethodcountdistin)
* [*Custom Aggregation Method*](#sec_CustomAggregationMethods) *–* a custom aggregation method that can be applied to expressions of a specified type
* [*Custom Aggregate*](#sec_CustomAggregates) *–* a dynamic property that can appear in an aggregate clause
* [*Groupable Property*](#sec_GroupableProperties) – a property whose values can be used to group entities or complex type instances for aggregation
* [*Hierarchy*](#sec_Hierarchies) – an arrangement of groupable properties whose values are represented as being “above”, “below”, or “at the same level as” one another
* *Expression* – derived from the commonExpr rule (see **[[OData-ABNF]](#ODataABNF)**)
* *Aggregatable Primitive Type* – a primitive type other than Edm.Stream or subtypes of Edm.Geography or Edm.Geometry
* *Aggregatable Expression* – an [expression](#ODataExpression) resulting in a value of an [aggregatable primitive type](#ODataAggregatablePrimitiveType)

## [Example Data Model](#sec_ExampleDataModel)

Example 2: The following diagram shows the terms defined in the section above applied to a simple model that is used throughout this document.



The Amount property in the Sales entity type is an aggregatable property, and the properties of the related entity types are groupable. These can be arranged in four hierarchies:

* Product hierarchy based on groupable properties of the Category and Product entity types
* Customer hierarchy based on Country and Customer
* Time hierarchy based on Year, Month and Date
* SalesOrganization based on the recursive association to itself

In the context of Online Analytical Processing (OLAP), this model might be described in terms of a Sales “cube” with an Amount “measure” and three “dimensions”. This document will avoid such terms, as they are heavily overloaded.

Query extensions and descriptive annotations can both be applied to normalized as well as partly or fully denormalized schemas.



Note that OData’s Entity Data Model (EDM) does not mandate a single storage model; it may be realized as a completely conceptual model whose data structure is calculated on-the-fly for each request. The actual "entity-relationship structure" of the model should be chosen to simplify understanding and querying data for the target audience of a service. Different target audiences may well require differently structured services on top of the same storage model.

## [Example Data](#sec_ExampleData)

Example 3: The following sample data will be used to further illustrate the capabilities introduced by this extension.



## [Example Use Cases](#sec_ExampleUseCases)

Example 4: In the example model, one prominent use case is the relation of customers to products. The first question that is likely to be asked is: “Which customers bought which products?”

This leads to the second more quantitative question: “Who bought how much of what?”

The answer to the second question typically is visualized as a cross-table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | Food |  |  | Non-Food |  |
|  |  |  |  | Sugar | Coffee |  | Paper |
| USA |  | USD | **14** | **2** | **12** | **5** | **5** |
|  |  Joe | USD | **6** | 2 | 4 | **1** | 1 |
|  |  Sue | USD | **8** |  | 8 | **4** | 4 |
| Netherlands |  | EUR | **2** | **2** |  | **3** | **3** |
|  |  Sue | EUR | **2** | 2 |  | **3** | 3 |

The data in this cross-table can be written down in a shape that more closely resembles the structure of the data model, leaving cells empty that have been aggregated away:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Customer/Country | Customer/Name | Product/Category/Name | Product/Name | Amount | Currency /Code |
| USA | Joe | Non-Food | Paper | 1 | USD |
| USA | Joe | Food | Sugar | 2 | USD |
| USA | Joe | Food | Coffee | 4 | USD |
| USA | Sue | Food | Coffee | 8 | USD |
| USA | Sue | Non-Food | Paper | 4 | USD |
| Netherlands | Sue | Food | Sugar | 2 | EUR |
| Netherlands | Sue | Non-Food | Paper | 3 | EUR |
| **USA** |  | **Food** | **Sugar** | **2** | **USD** |
| **USA** |  | **Food** | **Coffee** | **12** | **USD** |
| **USA** |  | **Non-Food** | **Paper** | **5** | **USD** |
| **Netherlands** |  | **Food** | **Sugar** | **2** | **EUR** |
| **Netherlands** |  | **Non-Food** | **Paper** | **1** | **EUR** |
| **USA** | **Joe** | **Food** |  | **6** | **USD** |
| **USA** | **Joe** | **Non-Food** |  | **1** | **USD** |
| **USA** | **Sue** | **Food** |  | **8** | **USD** |
| **USA** | **Sue** | **Non-Food** |  | **4** | **USD** |
| **Netherlands** | **Sue** | **Food** |  | **2** | **EUR** |
| **Netherlands** | **Sue** | **Non-Food** |  | **3** | **EUR** |
| **USA** |  | **Food** |  | **14** | **USD** |
| **USA** |  | **Non-Food** |  | **5** | **USD** |
| **Netherlands** |  | **Food** |  | **2** | **EUR** |
| **Netherlands** |  | **Non-Food** |  | **3** | **EUR** |

Note that this result contains seven fully qualified aggregate values, plus fifteen rollup rows with subtotal values, shown in bold.

# [System Query Option $apply](#sec_SystemQueryOptionapply)

Aggregation behavior on a collection of instances is triggered using the query option $apply. It takes a sequence of set transformations, separated by forward slashes to express that they are consecutively applied, i.e. the result of each transformation is the input to the next transformation. This is consistent with the use of service-defined bindable and composable functions in path segments.

Unless otherwise noted, each set transformation:

* preserves the structure of the input type, so the structure of the result fits into the data model of the service.
* does not necessarily preserve the number of instances in the result, as this will typically differ from the number of instances in the input set.
* does not necessarily guarantee that all properties of the result instances have a well-defined value.

So the actual (or relevant) structure of each intermediary result will often resemble a projection of the original data model that could also have been formed using the standard system query options $expand and $select defined in [**[OData-Protocol]**](#ODataProtocol), with dynamic properties representing the aggregate values. The parameters of set transformations allow specifying how the result instances are constructed from the input instances. This especially means that all property paths relate to the structure of the immediate input set.

The set transformations defined by this extension are

* [aggregate](#sec_Transformationaggregate)
* [topcount](#sec_Transformationtopcount)
* [topsum](#sec_Transformationtopsum)
* [toppercent](#sec_Transformationtoppercent)
* [bottomcount](#sec_Transformationbottomcount)
* [bottomsum](#sec_Transformationbottomsum)
* [bottompercent](#sec_Transformationbottompercent)
* [identity](#sec_Transformationidentity)
* [concat](#sec_Transformationconcat)
* [groupby](#sec_Transformationgroupby)
* [filter](#sec_Transformationfilter)
* [expand](#sec_Transformationexpand)
* [search](#sec_Transformationsearch)
* [compute](#sec_Transformationcompute)

Service-defined bound functions that take a collection as their binding parameter MAY be used as set transformations within $apply if the type of the binding parameter matches the type of the result set of the preceding transformation. If it returns a collection, further transformations can follow the bound function. The parameter syntax for bound function segments is identical to the parameter syntax for bound functions in resource path segments or $filter expressions. See section 7.6 for an example.

If a data service that supports $apply does not support it on the collection identified by the request resource path, it MUST fail with 501 Not Implemented and a meaningful human-readable error message.

The representation of the aggregated result set as response payload is described in section 4. The aggregated entities returned in the request examples in the following sections are transient and represented in the OData 4.01 JSON format that allows to omit the id control information for transient entities. For further brevity, the examples also leave out the type control information required to disambiguate the type of represented dynamic property values.

## [Transformation aggregate](#sec_Transformationaggregate)

The aggregate transformation takes a comma-separated list of one or more *aggregate expressions* as parameters and returns a result set with a single instance, representing the aggregated value for all instances in the input set.

An aggregate expression may be:

* an [aggregatable expression](#ODataAggregatableExpression) on the input set with a specified [aggregation method](#sec_AggregationMethods),
* an [aggregatable property](#ODataAggregatableProperty) that can be reached via a path consisting of a sequence of navigation properties, complex properties and complex collection properties, with a specified [aggregation method](#sec_AggregationMethods),
* a [custom aggregate](#sec_CustomAggregates) that can be reached via a path consisting of a sequence of navigation properties, complex properties and complex collection properties,
* any of the above, followed by a [from](#sec_Keywordfrom) expression,
* any of the above, enclosed in parentheses and prefixed with a navigation path to related entities, which serve as input set for the enclosed aggregate expression,
* [$count](#sec_VirtualPropertycount), which is a shortcut for the expression 1 with sum

Any aggregate expression that specifies an aggregation method MUST define an [alias](#sec_Keywordas) for the resulting aggregated value. The resulting instance contains one dynamic property per parameter representing the aggregated value across all instances within the input set. If paths are present, the corresponding navigation properties are implicitly expanded to make the properties part of the result representation.

The aggregate transformation affects the structure of the result set: An [aggregatable expression](#ODataAggregatableExpression) and a custom aggregate correspond to a dynamic property in a $select option. If the expression operates on related entities, the corresponding $select option would be nested in one $expand option for each navigation property in the navigation path.

### [Keyword as](#sec_Keywordas)

Aggregate expressions can define an alias using the as keyword, followed by a SimpleIdentifier (see [**[OData-CSDL, section 17.2]**](#ODataCSDL)).

The alias will introduce a dynamic property in the aggregated result set. The introduced dynamic property is added to the type containing the original expression or custom aggregate. The alias MUST NOT collide with names of declared properties, custom aggregates, or other aliases in that type.

When an [aggregation method](#sec_AggregationMethods) is specified, an alias MUST be applied to the expression.

Example 5:

GET ~/Sales?$apply=aggregate(Amount with sum as Total,Amount with max as MxA)

results in

{

 "@odata.context": "$metadata#Sales(Total,MxA)",

 "value": [
 { "Total": 24, "MxA": 8, ... }
 ]

}

Example 6:

GET ~/Sales?$apply=aggregate(Amount mul Product/TaxRate with sum as Tax)

results in

{

 "@odata.context": "$metadata#Sales(Tax)",

 "value": [
 { "Tax": 2.08 }

 ]

}

If the aggregate expression is to be evaluated on related entities, the expression and its alias MUST be enclosed in parentheses and prefixed with the navigation path to the related entities.

Example 7:

GET ~/Products?$apply=aggregate(Sales(Amount mul Product/TaxRate with sum as Tax))

results in

{

 "@odata.context": "$metadata#Products(Sales(Tax))",

 "value": [
 { "Sales": [ { "Tax": 2.08 } ] }
 ]

}

An alias affects the structure of the result set: each alias corresponds to a dynamic property in a $select option. If the aggregate expression with the alias operates on related entities, this dynamic property is nested in an $expand option for each navigation property in the path to the related entities.

### [Keyword with](#sec_Keywordwith)

The keyword with is used to apply an [aggregation method](#sec_AggregationMethods) to an [aggregatable expression](#ODataAggregatableExpression). The expression being aggregated is followed by the keyword with, followed by the name of the aggregation method to apply, followed by the keyword [as](#sec_Keywordas) and an alias.

### [Aggregation Methods](#sec_AggregationMethods)

Values can be aggregated using the standard aggregation methods [sum](#sec_StandardAggregationMethodsum), [min](#sec_StandardAggregationMethodmin), [max](#sec_StandardAggregationMethodmax), [average](#sec_StandardAggregationMethodaverage), and [countdistinct](#sec_StandardAggregationMethodcountdistin), or with [custom aggregation methods](#sec_CustomAggregationMethods) defined by the service. Aggregate expressions containing an aggregation method MUST define an [alias](#sec_Keywordas) for the resulting aggregate value.

#### [Standard Aggregation Method sum](#sec_StandardAggregationMethodsum)

The standard aggregation method sum can be applied to numeric values to return the sum of the non-null values, or null if there are no non-null values or the input set is empty. The provider MUST choose a single type for the property across all instances of that type in the result that is capable of representing the aggregated values. This may require a larger integer type, Edm.Decimal with sufficient Precision and Scale, or Edm.Double.

Example 8:

GET ~/Sales?$apply=aggregate(Amount with sum as Total)

results in

{

 "@odata.context": "$metadata#Sales(Total)",

 "value": [
 { "Total": 24, ... }
 ]

}

#### [Standard Aggregation Method min](#sec_StandardAggregationMethodmin)

The standard aggregation method min can be applied to values with a totally ordered domain to return the smallest of the non-null values, or null if there are no non-null values or the input set is empty.

The result property will have the same type as the input property.

Example 9:

GET ~/Sales?$apply=aggregate(Amount with min as MinAmount)

results in

{

 "@odata.context": "$metadata#Sales(MinAmount)",

 "value": [
 { "MinAmount": 1, ... }
 ]

}

#### [Standard Aggregation Method max](#sec_StandardAggregationMethodmax)

The standard aggregation method max can be applied to values with a totally ordered domain to return the largest of the non-null values, or null if there are no non-null values or the input set is empty.

The result property will have the same type as the input property

Example 10:

GET ~/Sales?$apply=aggregate(Amount with max as MaxAmount)

results in

{

 "@odata.context": "$metadata#Sales(MaxAmount)",

 "value": [
 { "MaxAmount": 8, ... }
 ]

}

#### [Standard Aggregation Method average](#sec_StandardAggregationMethodaverage)

The standard aggregation method average can be applied to numeric values to return the sum of the non-null values divided by the count of the non-null values, or null if there are no non-null values or the input set is empty.

The provider MUST choose a single type for the property across all instances of that type in the result that is capable of representing the aggregated values; either Edm.Double or Edm.Decimal with sufficient Precision and Scale.

Example 11:

GET ~/Sales?$apply=aggregate(Amount with average as AverageAmount)

results in

{

 "@odata.context": "$metadata#Sales(AverageAmount)",

 "value": [
 { "AverageAmount": 3.0 }
 ]

}

#### [Standard Aggregation Method countdistinct](#sec_StandardAggregationMethodcountdistin)

The aggregation method countdistinct counts the distinct values, omitting any null values. For navigation properties, it counts the distinct entities in the union of all entities related to entities in the input set. For collection-valued primitive properties, it counts the distinct items in the union of all collection values in the input set.

The result property MUST have type Edm.Decimal with Scale="0" and sufficient Precision.

Example 12:

GET ~/Sales?$apply=aggregate(Product with countdistinct as DistinctProducts)

results in

{

 "@odata.context": "$metadata#Sales(DistinctProducts)",

 "value": [
 { "DistinctProducts": 3, ... }

 ]

}

The number of instances in the input set can be counted with the aggregate expression [$count](#sec_VirtualPropertycount).

#### [Custom Aggregation Methods](#sec_CustomAggregationMethods)

Services can define custom aggregation methods if the functionality offered by the standard aggregation methods is not sufficient for the intended consumers.

Custom aggregation methods MUST use a namespace-qualified name (see [**[OData-ABNF]**](#ODataABNF)), i.e. contain at least one dot. Dot-less names are reserved for future versions of this specification.

Example 13: custom aggregation methods that concatenates distinct string values separated by commas

GET ~/Sales?$apply=groupby((Customer/Country),

 aggregate(Amount with sum as Total,

 Product/Name with Custom.concat as ProductNames))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country),Total,ProductNames)",

 "value": [
 { "Customer": { "Country": "Netherlands" },

 "Total": 5, ProductNames: "Paper,Sugar", ... },
 { "Customer": { "Country": "USA" },

 "Total": 19, ProductNames: "Coffee,Paper,Sugar", ... }
 ]
}

### [Keyword from](#sec_Keywordfrom)

The from keyword gives control over the order of aggregation across properties that are not part of the result structure and over the aggregation methods applied in every step.

Instead of applying a single aggregation method for calculating the value of an aggregate expression across all properties not included in the result structure, other aggregation methods to be applied when aggregating away certain properties MAY be specified using the from keyword, followed by a property path of a groupable property. Each groupable property MUST be followed by a with clause unless the aggregate expression is a custom aggregate, in which case the provider-defined behavior of the custom aggregate is used:

 from [ with ]

 …

from [ with ]

as

If the from keyword is used, an alias MUST be introduced.

If the from keyword is present, first the aggregation method determined by the aggregate expression is used to aggregate away properties that are not mentioned in a from clause and are not [grouping properties](#sec_Transformationgroupby).

Then consecutively properties not part of the result are aggregated away in the order of the from clauses and using the method specified by the from clause.

More formally, the calculation of aggregate with the from keyword is equivalent with a list of set transformations:

groupby((, …, ),

 aggregate( as ))

/groupby((, …, ),

 aggregate( with as ))

…

/groupby((),

 aggregate( with as ))

/aggregate( with as ))

The order of from clauses has to be compatible with hierarchies referenced from a [leveled hierarchy annotation](#sec_LeveledHierarchy) or specified as an unnamed hierarchy in [groupby with rollup](#sec_Groupingwithrollupandall): lower nodes in a hierarchy need to be mentioned before higher nodes in the same hierarchy. Properties not belonging to any hierarchy can appear at any point in the from clause.

If an aggregate transformation has more than one aggregate expression as parameter that uses the from keyword, then all properties mentioned in from clauses must appear in the same order in all aggregate expressions.

Example 14:

GET ~/Sales?$apply=aggregate(Amount with sum from Time with average

 as DailyAverage)

is equivalent to

GET ~/Sales?$apply=groupby((Time),aggregate(Amount with sum as Total))
 /aggregate(Total with average as DailyAverage)

and results in the average sales volume per day

{

 "@odata.context": "$metadata#Sales(DailyAverage)",

 "value": [
 { "DailyAverage": 3.428571428571429 }
 ]

}

### [Aggregate Expression $count](#sec_VirtualPropertycount)

The value of the aggregate expression $count is the number of instances in the input set. It MUST always specify an [alias](#sec_Keywordas) and MUST NOT specify an [aggregation method](#sec_AggregationMethods).

The result property will have type Edm.Int64.

Example 15:

GET ~/Sales?$apply=aggregate($count as SalesCount)

results in

{

 "@odata.context": "$metadata#Sales(SalesCount)",

 "value": [
 { "SalesCount": 8, ... }
 ]

}

## [Transformation topcount](#sec_Transformationtopcount)

The topcount transformation takes two parameters.

The first parameter specifies the number of instances to return in the transformed set. It MUST be an [expression](#ODataExpression) that can be evaluated on the set level and MUST result in a positive integer.

The second parameter specifies the value by which the instances are compared for determining the result set. It MUST be an [expression](#ODataExpression) that can be evaluated on instances of the input set and MUST result in a value of a primitive type whose values are totally ordered.

The transformation retains the number of instances specified by the first parameter that have the highest values specified by the second parameter. It does not change the order of the instances in the input set.

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.

Example 16:

GET ~/Sales?$apply=topcount(2,Amount)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 3, "Amount": 4, ... },
 { "ID": 4, "Amount": 8, ... }
 ]

}

The result set of topcount has the same structure as the input set.

## [Transformation topsum](#sec_Transformationtopsum)

The topsum transformation takes two parameters.

The first parameter indirectly specifies the number of instances to return in the transformed set. It MUST be an [expression](#ODataExpression) that can be evaluated on the set level and MUST result in a number.

The second parameter specifies the value by which the instances are compared for determining the result set. It MUST be an [aggregatable expression](#ODataAggregatableExpression) that can be evaluated on instances of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of instances that have the highest values specified by the second parameter and whose sum of these values is equal to or greater than the value specified by the first parameter. It does not change the order of the instances in the input set.

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.

Example 17:

GET ~/Sales?$apply=topsum(15,Amount)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 3, "Amount": 4, ... },
 { "ID": 4, "Amount": 8, ... },
 { "ID": 5, "Amount": 4, ... }
 ]

}

The result set of topsum has the same structure as the input set.

## [Transformation toppercent](#sec_Transformationtoppercent)

The toppercent transformation takes two parameters.

The first parameter specifies the number of instances to return in the transformed set. It MUST be an [expression](#ODataExpression) that can be evaluated on the set level and MUST result in a positive number less than or equal to 100.

The second parameter specifies the value by which the instances are compared for determining the result set. It MUST be an [aggregatable expression](#ODataAggregatableExpression) that can be evaluated on instances of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of instances that have the highest values specified by the second parameter and whose cumulative total is equal to or greater than the percentage of the cumulative total of all instances in the input set specified by the first parameter. It does not change the order of the instances in the input set.

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.

Example 18:

GET ~/Sales?$apply=toppercent(50,Amount)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 3, "Amount": 4, ... },
 { "ID": 4, "Amount": 8, ... }
 ]

}

The result set of toppercent has the same structure as the input set.

## [Transformation bottomcount](#sec_Transformationbottomcount)

The bottomcount transformation takes two parameters.

The first parameter specifies the number of instances to return in the transformed set. It MUST be an [expression](#ODataExpression) that can be evaluated on the set level and MUST result in a positive integer.

The second parameter specifies the value by which the instances are compared for determining the result set. It MUST be an [expression](#ODataExpression) that can be evaluated on instances of the input set and MUST result in a value of a primitive type whose values are totally ordered.

The transformation retains the number of instances specified by the first parameter that have the lowest values specified by the second parameter. It does not change the order of the instances in the input set.

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.

Example 19:

GET ~/Sales?$apply=bottomcount(2,Amount)

results in

{

 "@odata.context": "$metadata#Sales"

 "value": [
 { "ID": 1, "Amount": 1, ... },
 { "ID": 7, "Amount": 1, ... }
 ]

}

The result set of bottomcount has the same structure as the input set.

## [Transformation bottomsum](#sec_Transformationbottomsum)

The bottomsum transformation takes two parameters.

The first parameter indirectly specifies the number of instances to return in the transformed set. It MUST be an [expression](#ODataExpression) that can be evaluated on the set level and MUST result in a number.

The second parameter specifies the value by which the instances are compared for determining the result set. It MUST be an [aggregatable expression](#ODataAggregatableExpression) that can be evaluated on instances of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of instances that have the lowest values specified by the second parameter and whose sum of these values is equal to or greater than the value specified by the first parameter. It does not change the order of the instances in the input set.

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.

Example 20:

GET ~/Sales?$apply=bottomsum(7,Amount)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 2, "Amount": 2, ... },
 { "ID": 6, "Amount": 2, ... },
 { "ID": 7, "Amount": 1, ... },
 { "ID": 8, "Amount": 2, ... }
 ]

}

The result set of bottomsum has the same structure as the input set.

## [Transformation bottompercent](#sec_Transformationbottompercent)

The bottompercent transformation takes two parameters.

The first parameter indirectly specifies the number of instances to return in the transformed set. It MUST be an [expression](#ODataExpression) that can be evaluated on the set level and MUST result in a positive number less than or equal to 100.

The second parameter specifies the value by which the instances are compared for determining the result set. It MUST be an [aggregatable expression](#ODataAggregatableExpression) that can be evaluated on instances of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of instances that have the lowest values specified by the second parameter and whose cumulative total is equal to or greater than the percentage of the cumulative total of all instances in the input set specified by the first parameter. It does not change the order of the instances in the input set.

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.

Example 21:

GET ~/Sales?$apply=bottompercent(50,Amount)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 1, "Amount": 1, ... },
 { "ID": 2, "Amount": 2, ... },
 { "ID": 5, "Amount": 4, ... },
 { "ID": 6, "Amount": 2, ... },
 { "ID": 7, "Amount": 1, ... },
 { "ID": 8, "Amount": 2, ... }
 ]

}

The result set of bottompercent has the same structure as the input set.

## [Transformation identity](#sec_Transformationidentity)

The identity transformation returns its input set in unchanged order.

Example 22:

GET ~/Sales?$apply=identity

## [Transformation concat](#sec_Transformationconcat)

The concat transformation takes two or more parameters, each of which is a sequence of set transformations.

It applies each transformation sequence to the input set and concatenates the intermediate result sets in the order of the parameters into the result set, preserving the ordering of the individual result sets as well as the structure of each result instance, potentially leading to an inhomogeneously structured result set.

Example 23:

GET ~/Sales?$apply=concat(topcount(2,Amount),
 aggregate(Amount))

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 4, "Amount": 8, ... },
 { "ID": 3, "Amount": 4, ... },
 { "@odata.context": "$metadata#Sales(Amount)/$entity", "Amount": 24, ... }
 ]

}

Note that two Sales entities with the second highest amount 4 exist in the input set; the entity with ID 3 is included in the result, because the service chose to use the ID property for imposing a stable ordering.

The result set of concat has a mixed form consisting of the structures imposed by the two transformation sequences.

## [Transformation groupby](#sec_Transformationgroupby)

The groupby transformation takes one or two parameters and

1. Splits the initial set into subsets where all instances in a subset have the same values for the grouping properties specified in the first parameter,
2. Applies set transformations to each subset according to the second parameter, resulting in a new set of potentially different structure and cardinality,
3. Ensures that the instances in the result set contain all grouping properties with the correct values for the group,
4. Concatenates the intermediate result sets into one result set.

### [Simple Grouping](#sec_SimpleGrouping)

In its simplest form the first parameter of groupby specifies the *grouping properties*, a comma-separated list of one or more single-valued property paths (paths ending in a single-valued primitive, complex, or navigation property) that is enclosed in parentheses. A path may consist of a sequence of navigation properties, complex properties and complex collection properties. The syntax groupby((p/q), T) where the navigation property path p is multi-valued and q single-valued is kept for backwards compatibility only, this means the same as expand(p, groupby((q), T)). The same property path SHOULD NOT appear more than once; redundant property paths MAY be considered valid, but MUST NOT alter the meaning of the request. If the property path leads to a single-valued navigation property, this means grouping by the entity-id of the related entities.

The optional second parameter is a list of set transformations, separated by forward slashes to express that they are consecutively applied. Transformations may take into account the grouping properties for producing their result, e.g. aggregate removes properties that are used neither for grouping nor for aggregation.

If the service is unable to group by same values for any of the specified properties, it MUST reject the request with an error response. It MUST NOT apply any implicit rules to group instances indirectly by another property related to it in some way.

Example 24:

GET ~/Sales?$apply=groupby((Customer/Country,Product/Name),
 aggregate(Amount with sum as Total))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country),Product(Name),Total)",

 "value": [
 { "Customer": { "Country" : "Netherlands" },
 "Product": { "Name": "Paper" }, "Total": 3, ... },
 { "Customer": { "Country": "Netherlands" },
 "Product": { "Name": "Sugar" }, "Total": 2, ... },
 { "Customer": { "Country: "USA" },
 "Product": { "Name": "Coffee" }, "Total": 12, ... },
 { "Customer": { "Country: "USA" },
 "Product": { "Name": "Paper" },"Total": 5, ... },
 { "Customer": { "Country: "USA" },

 "Product": { "Name": "Sugar" }, "Total": 2, ... }

 ]

}

The second parameter can be omitted to request distinct value combinations of the grouping properties.

Example 25:

GET ~/Sales?$apply=groupby((Product/Name,Amount))

and result in

{

 "@odata.context": "$metadata#Sales(Product(Name),Amount)",

 "value": [
 { "Product": { "Name": "Coffee" }, "Amount": 4 },
 { "Product": { "Name": "Coffee" }, "Amount": 8 },
 { "Product": { "Name": "Paper" }, "Amount": 1 },
 { "Product": { "Name": "Paper" }, "Amount": 2 },
 { "Product": { "Name": "Paper" }, "Amount": 4 },
 { "Product": { "Name": "Sugar" }, "Amount": 2 }
 ]

}

Note that the result has the same structure, but not the same content as

GET ~/Sales?$expand=Product($select=Name)&$select=Amount

A groupby transformation affects the structure of the result set similar to $select where each grouping property corresponds to an item in a $select clause.

Grouping by a navigation property is treated as grouping by the references of the navigation targets and adds the canonical URL of the target entities as deferred representation of the navigation property to the result structure, which then can be expanded and projected partially away using the standard query options $expand and $select.

Grouping properties that specify navigation properties are automatically expanded, and the specified properties of that navigation property correspond to properties specified in a $select expand option on the expanded navigation property. The set transformations specified in the second parameter of groupby further affect the structure as described for each transformation; for example, the [aggregate](#sec_Transformationaggregate) transformation adds properties for each aggregate expression.

### [Grouping with rollup and $all](#sec_Groupingwithrollupandall)

The rollup grouping operator allows requesting additional levels of aggregation in addition to the most granular level defined by the grouping properties. It can be used instead of a property path in the first parameter of groupby.

The rollup grouping operator has two overloads, depending on the number of parameters.

If used with one parameter, the parameter MUST be the value of the Qualifier attribute of an annotation with term [LeveledHierarchy](#sec_LeveledHierarchy) prefixed with the navigation path leading to the annotated entity type. This named hierarchy is used for grouping instances.

If used with two or more parameters, it defines an unnamed leveled hierarchy. The first parameter is the root of the hierarchy defining the coarsest granularity and MUST either be a single-valued property path or the virtual property $all. The other parameters MUST be singe-valued property paths and define consecutively finer-grained levels of the hierarchy. This unnamed hierarchy is used for grouping instances.

After resolving named hierarchies, the same property path MUST NOT appear more than once.

Grouping with rollup is processed for leveled hierarchies using the following equivalence relationships, in which *pn* is a property path, *T* is a transformation, the ellipsis stands in for zero or more property paths, and *R* stands in for zero or more rollup operators or property paths:

* groupby((rollup(*p*1,…,*pn*-1,*pn*),*R*),*T*) is equivalent to
concat(groupby((*p*1,…,*pn*-1,,*R*,*T*),groupby((rollup(*p*1,…,*pn*-1),*R*),*T*))
* groupby((rollup(*p*1,*p*2),*R*),*T*) is equivalent to
concat(groupby((*p*1,*p*2,*R*),*T*),groupby((,*R*),*T*))
* groupby((rollup($all, *p*1),*R*),*T*) is equivalent to
concat(groupby((*p*1,*R*),*T*),groupby((*R*),*T*))
* groupby((rollup($all,*p*1)),*T*) is equivalent to
concat(groupby((*p*1),*T*),*T*)

Loosely speaking groupby with rollup splits the input set into groups using all grouping properties, then removes the last property from one of the hierarchies and splits it again using the remaining grouping properties. This is repeated until all of the hierarchies have been used up.

Example 26: rolling up two hierarchies, the first with two levels, the second with three levels:

(rollup(*p*1,1,*p*1,2),rollup(*p*2,1,*p*2,2,*p*2,3))

will result in the six groupings

 (*p*1,1,*p*1,2, *p*2,1,*p*2,2,*p*2,3)

 (*p*1,1,*p*1,2, *p*2,1,*p*2,2)

 (*p*1,1,*p*1,2, *p*2,1)

 (*p*1,1, *p*2,1,*p*2,2,*p*2,3)

 (*p*1,1, *p*2,1,*p*2,2)

 (*p*1,1, *p*2,1)

Note that rollup stops one level earlier than GROUP BY ROLLUP in TSQL, see [TSQL ROLLUP], unless the virtual property $all is used as the hierarchy root level. Loosely speaking the root level is never rolled up.

Ordering of rollup instances within detail instances is up to the service if no $orderby is given, otherwise at the position determined by $orderby.

Example 27: answering the second question in section 2.4

GET ~/Sales?$apply=groupby((rollup(Customer/Country,Customer/Name),
 rollup(Product/Category/Name,Product/Name),
 Currency/Code),
 aggregate(Amount with sum as Total))

results in seven entities for the finest grouping level

{

"@odata.context":"$metadata#Sales(Customer(Country,Name),Product(Category(Name),Name),Total,Currency(Code))",

 "value": [
 { "Customer": { "Country": "USA", "Name": "Joe" },
 "Product": { "Category": { "Name": "Non-Food" }, "Name": "Paper" },
 "Total": 1, "Currency": { "Code": "USD" }, ...
 },
 ...

plus additional fifteen rollup entities for subtotals: five without customer name

 { "Customer": { "Country": "USA" },
 "Product": { "Category": { "Name": "Food" }, "Name": "Sugar" },
 "Total": 2, "Currency": { "Code": "USD" }, ...
 },
 ...

six without product name

 { "Customer": { "Country": "USA", "Name": "Joe" },
 "Product": { "Category": { "Name": "Food" } },
 "Total": 6, "Currency": { "Code": "USD" }, ...
 },

 ...

and four with neither customer nor product name

 { "Customer": { "Country": "USA" },
 "Product": { "Category": { "Name": "Food" } },
 "Total": 14, "Currency": { "Code": "USD" }, ...
 },
 ...
 ]

}

Note that the absence of one or more properties of the result structure imposed by the surrounding OData context allows distinguishing rollup entities from other entities.

## [Transformation filter](#sec_Transformationfilter)

The filter transformation takes a Boolean expression that could also be passed as a $filter system query option to its input set and returns all instances for which this expression evaluates to true.

Example 28:

GET ~/Sales?$apply=filter(Amount gt 3)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 3, "Amount": 4, ... },
 { "ID": 4, "Amount": 8, ... },
 { "ID": 5, "Amount": 4, ... }
 ]

}

The result set of filter has the same structure as the input set.

## [Transformation expand](#sec_Transformationexpand)

The expand transformation takes a navigation property path that could also be passed as a $expand system query option as its first parameter. The second parameter specifies a transformation sequence that will be applied to the related entities. An arbitrary number of expand transformations can be passed as additional parameters to achieve multi-level expansion.

For a property path s=p/q, where p is a navigation property and q is a navigation property path, expand(s, T) is recursively defined as expand(p, expand(q, T)). Additional parameters to expand with further expand transformations like expand(p, T, expand(q, U),…) are recursively defined as expand(p,T)/expand(p/q,U)/…

For a navigation property p and transformation sequence T, the result set of expand(p, T) is determined by the following steps.:

1. Verify constraints:
	1. Ensure that p is addressable for all instances of the input set and its target(s) can be dereferenced, otherwise reject the request.
	2. If p is single-valued, ensure that T is either an expand or an identity transformation, otherwise reject the request.
2. Start with an empty result set.
3. If p is collection-valued: For each entity e in the input set:
	1. Form a new "nested input set" from the target of the navigation property p.
	2. Apply transformation T to the nested input set from step (a).
	3. Add an instance to the result set that is a clone of e with an additional dynamic collection-valued property with name p, whose value is the intermediate result from step (b).
4. If p is single-valued: For each entity e in the input set: (Q: what if null?)
	1. Form a new "singleton input set" from the target of the navigation property p.
	2. Apply the transformation T to the singleton input set from step (a). The result set is also a singleton.
	3. Add an instance to the result set that is a clone of e with an additional dynamic complex property with name p, whose value is the singleton result from step (b).

Example 29:

GET ~/Customers?$apply=expand(Sales,filter(Amount gt 3))

results in

{

 "@odata.context": "$metadata#Customers",

 "value": [
 { "ID": "C1", "Name": "Joe", "Country": "USA",
 "Sales": [{ "ID": 3, "Amount": 4, ... }]},
 { "ID": "C2", "Name": "Sue", "Country": "USA",
 "Sales": [{ "ID": 4, "Amount": 8, ... },
 { "ID": 5, "Amount": 4, ... }]},
 { "ID": "C3", "Name": "Sue", "Country": "Netherlands", "Sales": []},
 { "ID": "C4", "Name": "Luc", "Country": "France", "Sales": []}
 ]

}

The result has the same structure and content as

GET ~/Customers?$expand=Sales($filter=Amount gt 3)

An expand transformation affects the structure of the result set in the same way as an $expand option for the first parameter, with nested $expand options for the optional nested expand transformations.

Example 30: nested expand transformations

GET ~/Categories?$apply=expand(Products,expand(Sales,filter(Amount gt 3)))

results in

{

 "@odata.context": "$metadata#Categories",

 "value": [

 { "ID": "PG1", "Name": "Food",

 "Products": [
 { "ID": "P1", "Name": "Sugar", "Color": "White", "Sales": [] },
 { "ID": "P2", "Name": "Coffee", "Color": "Brown",

 "Sales": [ { "ID": 3, "Amount": 4, ... },
 { "ID": 4, "Amount": 8, ... } ] }

 ]

 },
 { "ID": "PG2", "Name": "Non-Food",

 "Products": [
 { "ID": "P3", "Name": "Paper", "Color": "White",

 "Sales": [ { "ID": 5, "Amount": 4, ... } ] },
 { "ID": "P4", "Name": "Pencil", "Color": "Black", "Sales": [] }

 ]

 }
 ]

}

## [Transformation search](#sec_Transformationsearch)

The search transformation takes a search expression that could also be passed as a $search system query option to its input set and returns all entities that match this search expression.

Example 31: assuming that free-text search on Sales takes the related product name into account,

GET ~/Sales?$apply=search(coffee)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 3, "Amount": 4, ... },
 { "ID": 4, "Amount": 8, ... }
 ]

}

The result set of search has the same structure as the input set.

## [Transformation compute](#sec_Transformationcompute)

The compute transformation takes a comma-separated list of one or more *compute expressions* as parameters.

A compute expression is an [aggregatable expression](#ODataAggregatableExpression) on the input set, followed by the as keyword, followed by a SimpleIdentifier (see [**[OData-CSDL, section 17.2]**](#ODataCSDL)), called an alias. This alias MUST NOT collide with names of properties in the input set or with other aliases introduced in the same compute transformation.

The result set is constructed by copying the instances of the input set and adding one dynamic property per compute expression to each instance of the input set. The name of the added property is the alias following the as keyword. The value of the property is the value of the compute expression evaluated on that instance. The type of the property is determined by the rules for evaluating $filter expressions and numeric promotion defined in [**[OData-URL]**](#ODataURL).

The values of properties copied from the input set are not changed, nor is the order of instances changed.

Example 32:

GET ~/Sales?$apply=compute(Amount mul Product/TaxRate as Tax)

results in

{

 "@odata.context": "$metadata#Sales",

 "value": [
 { "ID": 1, ..., "Amount": 1, "Tax": 0.14 },

 { "ID": 2, ..., "Amount": 2, "Tax": 0.12 },

 { "ID": 3, ..., "Amount": 4, "Tax": 0.24 },

 { "ID": 4, ..., "Amount": 8, "Tax": 0.48 },

 { "ID": 5, ..., "Amount": 4, "Tax": 0.56 },

 { "ID": 6, ..., "Amount": 2, "Tax": 0.12 },

 { "ID": 7, ..., "Amount": 1, "Tax": 0.14 },

 { "ID": 8, ..., "Amount": 2, "Tax": 0.28 }

 ]

}

## [Transformation orderby](#sec_Transformationfilter)

The orderby transformation takes an expression that could also be passed as a $orderby system query option and returns the instances of the input set in the same order $orderby would produce for the given expression. A service supporting this transformation MUST at least offer sorting by property paths, including dynamic properties, with both suffixes asc and desc.

Example 33:

GET ~/Sales?$apply=groupby((Product/Name),

 aggregate(Amount with sum as Total))

 /orderby(Total desc)

results in

{

 "@odata.context": "$metadata#Sales(Product(Name),Total)",

 "value": [
 { "Product": { "Name": "Coffee" }, "Total": 12, ... },
 { "Product": { "Name": "Paper" }, "Total": 8, ... },
 { "Product": { "Name": "Sugar" }, "Total": 4, ... }
 ]

}

The result set of orderby has the same structure as the input set.

## [Transformation top](#sec_Transformationfilter)

The top transformation takes a non-negative integer n as argument and returns the first n entities of the input set, if the input set contains more than n entities, otherwise the entire input set. The entities in the result are in the same order as they occur in the input set.

The transformations preceding top in the sequence must first produce a certain order among the entities, and then keep this order stable for the input set of top.

Example 34:

GET ~/Sales?$apply=orderby(Customer/Name desc)/top(2)

results in

{

 "@odata.context": "$metadata#Sales"

 "value": [
 { "ID": 4, "Amount": 8, ... },
 { "ID": 5, "Amount": 4, ... }
 ]

}

## [Transformation skip](#sec_Transformationfilter)

The skip transformation takes a non-negative integer n as argument that excludes the first n entities of the input set from the result. It returns all remaining entities in the same order as they occur in the input set.

The transformations preceding skip in the sequence must first produce a certain order among the entities, and then keep this order stable for the input set of skip.

Example 35:

GET ~/Sales?$apply=orderby(Customer/Name desc)/skip(2)/top(2)

results in

{

 "@odata.context": "$metadata#Sales"

 "value": [
 { "ID": 6, "Amount": 2, ... },
 { "ID": 7, "Amount": 1, ... }
 ]

}

## [Transformation nest](#sec_Transformationfilter)

The nest transformation takes a comma-separated list of one or more nest expressions as parameters. A nest expression is a sequence of set transformations, followed by the as keyword, followed by a SimpleIdentifier (see **[[OData-CSDL, section 17.2]](#ODataCSDL)**) , called an alias. This alias MUST NOT collide with names of properties in the input set or with other aliases introduced in the same nest transformation.

The result set consists of a single entity having one dynamic property per nest expression. The name of the added property is the alias following the as keyword. The value of the property is the result collection of the transformation sequence in the nest expression applied to the input set.

Example 35:

GET Sales?$apply = groupby((Product/Category/ID),

 nest(groupby((Customer/ID)) as Customers))

results in

{

"@odata.context":"$metadata#Sales(Product(Category(ID)),Customers(Customer(ID)))",

 "value": [

 { "Product": { "Category": { "ID": "PG1"} },

 "Customers": [ { "Customer": { "ID": "C1"} },

 { "Customer": { "ID": "C2" } },

 { "Customer": { "ID": "C3" } } ] },

 { "Product": { "Category": { "ID": "PG2"} },

 "Customers": [ { "Customer": { "ID": "C1"} },

 { "Customer": { "ID": "C2" } },

 { "Customer": { "ID": "C3" } } ] }

 ]

}

## [Function isdefined](#sec_FilterFunctionisdefined)

Properties that are not explicitly mentioned in [aggregate](#sec_Transformationaggregate) or [groupby](#sec_Transformationgroupby) are considered to have been *aggregated away* and are treated as having the null value in $filter expressions.

The function isdefined can be used to determine whether a property has been aggregated away. It takes a single-valued property path as its only parameter and returns true if the property has a defined value for the aggregated entity. A property with a defined value can still have the null value; it can represent a grouping of null values, or an aggregation that results in a null value.

Example 36:Product has been aggregated away, causing an empty result

GET ~/Sales?$apply=aggregate(Amount with sum as Total)

 &$filter=isdefined(Product)

results in

{

 "@odata.context": "$metadata#Sales(Total)",

 "value": []

}

## Function aggregate

The aggregate function allows to use aggregated values in [expressions](#ODataExpression). It takes a single parameter accepting an aggregate-function expression and returns the aggregated value of type Edm.PrimitiveType as the result from applying the aggregate-function expression on an input collection provided by the context.

An aggregate-function expression offers the same capabilities as an aggregate expression defined for the aggregate transformation except that it does not support aggregation on related entities, and it never includes an alias for introducing a dynamic property containing the aggregated value.

The function can be applied in any context accepting an [expression](#ODataExpression). Within $apply, the function is applied as part of an argument expression to a transformation, and the input set of that transformation is the input collection for the function. Outside of $apply, the function may be applied in requests querying collections in an argument expression for a system query option and a lambda operator. Inside a system query option, the collection on which it operates is used as input collection for the function. This is also valid for nested query options used as arguments, like $filter in $expand. Inside a lambda predicate expression the input collection for the aggregate function depends on the path prefix used in the aggregateFunctionExpr (see **[[OData-Agg-ABNF]](#ODataAggABNF)**): For a path expression starting with the lambda variable the input collection is given by the target of the navigation path prepended to the lambda operator; for a path expression starting with $it the collection addressed by the resource path is the input collection; for other path expressions the input collection is given by the collection of instances at the origin of the navigation path prepended to the lambda operator.

Example 36:

GET ~/Products?$filter=Sales/any(p:aggregate(p/Amount with sum) gt 5)

results in

{

 "@odata.context": "$metadata#Product",

 "value": [

 { "ID": "P2", ... },

 { "ID": "P3", ... }

 ]

}

## Transformation join

The join transformation takes a property path as its first parameter. A path may consist of a sequence of navigation and complex properties. The optional second parameter specifies a transformation sequence.

For a property path s=p/q, where p is a property and q is a property path, join(s, T) is recursively defined as join(p, join(q, T)). If T is not given, it is set to identity. For a property p and transformation sequence T, the result set of join(p, T) is determined by the following steps.:

1. Ensure that p is addressable for all instances of the input set and if it is a navigation property that its target can be dereferenced, otherwise reject the request.
2. Start with an empty result set.
3. For each entity e in the input set:
	1. Form a new "nested input set" from the instance(s) identified by p.
	2. Apply transformation T to the nested input set from step (a).
	3. For each instance r in the intermediate result of step (b), add an instance to the result set that is a clone of e with an additional complex property with name p and value r.

## [Evaluating $apply](#sec_Evaluatingapply)

The new system query option [$apply](#sec_SystemQueryOptionapply) is evaluated first, then the other system query options are evaluated, if applicable, on the result of $apply in their normal order (see [**[OData-Protocol, section 11.2.1]**](#ODataProtocol)). If the result is a collection, $filter, $orderby, $expand and $select work as usual on properties that are defined on the output set after evaluating [$apply](#sec_SystemQueryOptionapply).

Properties that have been aggregated away in a result entity are not represented, even if the properties are listed in $select or $expand. In $filter they are treated as having the null value, and in $orderby as having a value that is even lower than null, i.e. instances for which a property has been rolled up appear before instances that have a null value for that property when ordering ascending. In subsequent transformations, they are treated like in $filter as having the null value.

On resource paths ending in /$count the system query option [$apply](#sec_SystemQueryOptionapply) is evaluated on the set identified by the resource path without the /$count segment, the result is the plain-text number of items in the result of [$apply](#sec_SystemQueryOptionapply). This is similar to the combination of /$count and $filter.

The $count system query option is evaluated after [$apply](#sec_SystemQueryOptionapply), the annotation @odata.count contains the number of items in the result of [$apply](#sec_SystemQueryOptionapply).

Providers MAY support $count, $top and $skip together with [rollup](#sec_Groupingwithrollupandall), in which case rollup instances are counted identically to detail instances, i.e. $skip=5 skips the first five instances, independently of whether some of them are rollup entities.

If a provider cannot satisfy a request using [$apply](#sec_SystemQueryOptionapply), it MUST respond with 501 Not Implemented and a human-readable error message.

## [Evaluating $apply as an Expand Option](#sec_EvaluatingapplyasanExpandOption)

The new system query option [$apply](#sec_SystemQueryOptionapply) can be used as an expand option to inline the result of aggregating related entities. The rules for [evaluating $apply](#sec_Evaluatingapply) are applied in the context of the expanded navigation, i.e. $apply is evaluated first, and other expand options on the same navigation property are evaluated on the result of $apply.

Example 37: products with aggregated sales:

GET Products?$expand=Sales($apply=aggregate(Amount with sum as Total))

results in

{

 "@odata.context":"$metadata#Products(Sales(Total))",

 "value": [
 { "Name": "Coffee", "Color": "Brown", "TaxRate": 0.06,

 "Sales": [ { "Total": 12, ... } ] },
 { "Name": "Paper", "Color": "White", "TaxRate": 0.14,

 "Sales": [ { "Total": 8, ... } ] },
 { "Name": "Pencil", "Color": "Black", "TaxRate": 0.14,

 "Sales": [ { "Total": null } ] },
 { "Name": "Sugar", "Color": "White", "TaxRate": 0.06,

 "Sales": [ { "Total": 4, ... } ] }
 ]

}

## [ABNF for Extended URL Conventions](#sec_ABNFforExtendedURLConventions)

The normative ABNF construction rules for this specification are defined in [**[OData-Agg-ABNF]**](#ODataAggABNF). They incrementally extend the rules defined in [**[OData-ABNF]**](#ODataABNF).

# [Representation of Aggregated Instances](#sec_RepresentationofAggregatedInstances)

Aggregated instances are based on the structure of the individual instances from which they have been calculated, so the structure of the results fits into the data model of the service.

Properties that have been aggregated away are not represented at all in the aggregated instances.

Dynamic properties introduced through an [alias](#sec_Keywordas), through transformation nest or with custom aggregates are represented as defined by the response format. In particular, this includes a type annotation as additional control information to disambiguate the type of the represented value where required by the response format.

Aggregated instances are logically instances of the declared type of the collection identified by the resource path of the request. If the resource path identifies a collection of entities, the aggregated instances are also entities. These aggregated entities can be transient or persistent. Transient entities don’t possess an edit link or read link, and in the JSON representation of OData 4.0 payloads are marked with "@odata.id": null, which can be omitted in 4.01 payloads, see [**[OData-JSON]**](#ODataJSON). Edit links or read links of persistent entities MUST encode the necessary information to re-retrieve that particular aggregate value. How the necessary information is exactly encoded is not part of this specification. Only the boundary conditions defined in [**[OData-Protocol]**](#ODataProtocol), sections 4.1 and 4.2 MUST be met.

The context URL describes the superset of properties occurring in entities of the aggregated result set.

Example 38: looking again to the sample request for getting sales amounts per product and country presented in section 3.10.1 (Example 24):

GET ~/Sales?$apply=groupby((Customer/Country,Product/Name),

 aggregate(Amount with sum as Total))

will return corresponding metadata as shown here for a single transient aggregated entity:

{

 "@odata.context":"$metadata#Sales(Customer(Country),Product(Name),Total)",

 "value": [
 { "Customer": { "Country": "Netherlands" },
 "Product": { "Name": "Paper" },
 "Total": 3, ...

 },
 ...
 ]

}

# [Cross-Joins and Aggregation](#sec_CrossJoinsandAggregation)

OData supports querying related entities through defining navigation properties in the data model. These navigation paths help guide simple consumers in understanding and navigating relationships.

In some cases, however, requests need to span entity sets with no predefined associations. Such requests can be sent to the special resource $crossjoin instead of an individual entity set. The cross join of a list of entity sets is the Cartesian product of the listed entity sets, represented as a collection of complex type instances that have a navigation property with cardinality to-one for each participating entity set, and queries across entity sets can be formulated using these navigation properties. See [**[OData-URL]**](#ODataURL) for details.

Where useful navigations exist it is beneficial to expose those as explicit navigation properties in the model, but the ability to pose queries that span entity sets not related by an association provides a mechanism for advanced consumers to use more flexible join conditions.

Example 39: if Sales had a string property ProductID instead of the navigation property Product, a “join” between Sales and Products could be accessed via the $crossjoin resource

GET ~/$crossjoin(Products,Sales)
 ?$expand=Products($select=Name),Sales($select=Amount)
 &$filter=Products/ID eq Sales/ProductID

results in

{

 "@odata.context": "$metadata#Collection(Edm.ComplexType)",

 "value": [
 { "Products": { "Name": "Paper" }, "Sales": { "Amount": 1 } },
 { "Products": { "Name": "Sugar" }, "Sales": { "Amount": 2 } },
 ...

 ]

}

Example 40: using the $crossjoin resource for aggregate queries

GET ~/$crossjoin(Products,Sales)
 ?$apply=filter(Products/ID eq Sales/ProductID)
 /groupby((Products/Name),

 aggregate(Sales(Amount with sum as Total)))

results in

{

 "@odata.context": "$metadata#Collection(Edm.ComplexType)",

 "value": [
 { "Products": { "Name": "Coffee" }, "Sales": { "Total": 12, ... } },
 { "Products": { "Name": "Paper" }, "Sales": { "Total": 8, ... } },
 { "Products": { "Name": "Sugar" }, "Sales": { "Total": 4, ... } }
 ]

}

The entity container may be annotated in the same way as entity sets to express which aggregate queries are supported, see section 6.

# [Vocabulary for Data Aggregation](#sec_VocabularyforDataAggregation)

The following terms are defined in the vocabulary for data aggregation [**[OData-VocAggr]**](#ODataVocAggr).

## [Aggregation Capabilities](#sec_AggregationCapabilities)

The term ApplySupported can be applied to an entity container or to structured types and describes the aggregation capabilities of the entity container or of collections of instances of the annotated structured types. If present, it implies that instances of the annotated structured type, or of structured types used in the annotated entity container, can contain dynamic properties as an effect of $apply even if they do not specify the OpenType attribute, see [**[OData-CSDL]**](#ODataCSDL). The term has a complex type with the following properties:

* The Transformations collection lists all supported set transformations. Allowed values are the names of the standard transformations introduced in section 3 and namespace-qualified names identifying a service-defined bindable function. If Transformations is omitted the server supports all transformations defined by this specification.
* The CustomAggregationMethods collection lists supported custom aggregation methods. Allowed values are namespace-qualified names identifying service-specific aggregation methods. If omitted, no custom aggregation methods are supported.
* Rollup specifies whether the service supports no rollup, only a single rollup hierarchy, or multiple rollup hierarchies in a [groupby](#sec_Transformationgroupby) transformation. If omitted, multiple rollup hierarchies are supported.
* PropertyRestrictions specifies whether all properties can be used in [groupby](#sec_Transformationgroupby) and [aggregate](#sec_Transformationaggregate). If not specified, or specified with a value of false, all properties can be grouped and aggregated. If specified with a value of true clients have to check which properties are tagged as [Groupable](#sec_GroupableProperties) or [Aggregatable](#sec_AggregatableProperties).

All properties of ApplySupported are optional, so it can be used as a tagging annotation to signal unlimited support of aggregation.

Example 41: an entity container supporting everything defined in this specification.

<EntityContainer Name="SalesData">

 <Annotation Term="Aggregation.ApplySupported" />

 ...

</EntityContainer>

## [Property Annotations](#sec_PropertyAnnotations)

### [Groupable Properties](#sec_GroupableProperties)

If a structured type is annotated with ApplySupported or used within an entity container that is annotated with ApplySupported, and the ApplySupported annotation has a value of true for PropertyRestrictions, only those properties that are annotated with the tagging term Groupable can be used in [groupby](#sec_Transformationgroupby).

### [Aggregatable Properties](#sec_AggregatableProperties)

If a structured type is annotated with ApplySupported or used within an entity container that is annotated with ApplySupported, and the ApplySupported annotation has a value of true for PropertyRestrictions, only those properties that are annotated with the tagging term Aggregatable can be used in [aggregate](#sec_Transformationaggregate).

### Supported Aggregation Methods

If a structured type is annotated with ApplySupported or used within an entity container that is annotated with ApplySupported, all aggregation methods can be applied to an aggregatable property by default unless it is annotated with SupportedAggregationMethods that lists the subset of those aggregation methods that the service supports for this property.

### Recommended Aggregation Method

If a structured type is annotated with ApplySupported or used within an entity container that is annotated with ApplySupported, the service may advertise a recommended aggregation method for an aggregatable property to the consumer by annotating it with RecommendedAggregationMethod.

### [Custom Aggregates](#sec_CustomAggregates)

The term CustomAggregate allows defining dynamic properties that can be used in [aggregate](#sec_Transformationaggregate). No assumptions can be made on how the values of these custom aggregates are calculated, and which input values are used.

When applied to a structured type, the annotation specifies custom aggregates that are available for collections of instances of that structured type. When applied to an entity container, the annotation specifies custom aggregates whose input set may span multiple entity sets within the container.

A custom aggregate is identified by the value of the Qualifier attribute when applying the term. The value of the Qualifier attribute is the name of the dynamic property. The name MUST NOT collide with the names of other custom aggregates of the same model element.

The value of the annotation is a string with the qualified name of a primitive type or type definition in scope that specifies the type returned by the custom aggregate.

If the custom aggregate is associated with a structured type, the value of the Qualifier attribute MAY be identical to the name of a declared property of the structured type. In this case, the value of the annotation MUST have the same value as the Type attribute of the declared property. This is typically done when the custom aggregate is used as a default aggregate for that property. In this case, the name refers to the custom aggregate within an aggregate expression without a [with](#sec_Keywordwith) clause, and to the property in all other cases.

If the custom aggregate is associated with an entity container, the value of the Qualifier attribute MUST NOT collide with the names of any entity sets defined in the entity container.

Example 42: Sales forecasts are modeled as a custom aggregate of the Sales entity type because it belongs there. For the budget, there is no appropriate structured type, so it is modeled as a custom aggregate of the SalesData entity container.

<Annotations Target="SalesModel.Sales">

 <Annotation Term="Aggregation.CustomAggregate" Qualifier="Forecast"

 String="Edm.Decimal" />

</Annotations>

<Annotations Target="SalesModel.SalesData">

 <Annotation Term="Aggregation.CustomAggregate" Qualifier="Budget"

 String="Edm.Decimal" />

</Annotations>

These custom aggregates can be used in the aggregate transformation:

GET ~/Sales?$apply=groupby((Time/Month),aggregate(Forecast))

and:

GET ~/$crossjoin(Time)?$apply=groupby((Time/Year),aggregate(Budget))

### [Context-Defining Properties](#sec_ContextDefiningProperties)

Sometimes the value of a property or custom aggregate is only well-defined within the context given by values of other properties, e.g. a postal code together with its country, or a monetary amount together with its currency unit. These context-defining properties can be listed with the term ContextDefiningProperties whose type is a collection of property paths.

If present, the context-defining properties SHOULD be used as grouping properties when aggregating the annotated property or custom aggregate, or alternatively be restricted to a single value by a pre-filter operation. Services MAY respond with 400 Bad Request if the context-defining properties are not sufficiently specified for calculating a meaningful aggregate value.

### [Annotation Example](#sec_AnnotationExample)

Example 43: This simplified Sales entity type has a single aggregatable property Amount whose context is defined by the Code property of the related Currency, and a custom aggregate Forecast with the same context. The Code property of Currency is groupable. All other properties are neither groupable nor aggregatable.

<EntityType Name="Currency">

 <Key>

 <PropertyRef Name="Code" />

 </Key>

 <Property Name="Code" Type="Edm.String">

 <Annotation Term="Aggregation.Groupable" />

 </Property>

 <Property Name="Name" Type="Edm.String">

 <Annotation Term="Core.IsLanguageDependent" />

 </Property>

</EntityType>

<EntityType Name="Sales">

 <Key>

 <PropertyRef Name="ID" />

 </Key>

 <Property Name="ID" Type="Edm.String" Nullable="false" />

 <Property Name="Amount" Type="Edm.Decimal" Scale="variable">

 <Annotation Term="Aggregation.Aggregateable" />

 <Annotation Term="Aggregation.ContextDefiningProperties">

 <Collection>

 <PropertyPath>Currency/Code</PropertyPath>

 </Collection>

 </Annotation>

 </Property>

 <NavigationProperty Name="Currency" Type="SalesModel.Currency"
 Nullable="false">

 <Annotation Term="Aggregation.Groupable" />

 </NavigationProperty>

 <Annotation Term="Aggregation.CustomAggregate" Qualifier="Forecast"

 String="Edm.Decimal">

 <Annotation Term="Aggregation.ContextDefiningProperties">

 <Collection>

 <PropertyPath>Currency/Code</PropertyPath>

 </Collection>

 </Annotation>

 </Annotation>

</EntityType>

<EntityContainer Name="SalesData">

 <Annotation Term="Aggregation.ApplySupported">

 <Record>

 <PropertyValue Property="PropertyRestrictions" Bool="true" />

 </Record>

 </Annotation>

 <EntitySet Name="Sales" EntityType="SalesModel.Sales" />

 <EntitySet Name="Currencies" EntityType="SalesModel.Currency" /> </EntityContainer>

## [Hierarchies](#sec_Hierarchies)

A hierarchy is an arrangement of groupable properties whose values are represented as being “above”, “below”, or “at the same level as” one another. A hierarchy can be *leveled* or *recursive.*

### [Leveled Hierarchy](#sec_LeveledHierarchy)

A *leveled hierarchy* has a fixed number of levels each of which is represented by a groupable property. The values of a lower-level property depend on the property value of the level above.

A leveled hierarchy of an entity type is described with the term LeveledHierarchy that lists the properties used to form the hierarchy.

The order of the collection is significant: it lists the properties representing the levels, starting with the root level (coarsest granularity) down to the lowest level of the hierarchy.

The term LeveledHierarchy can only be applied to entity types, and the applying Annotation element MUST specify the Qualifier attribute. The value of the Qualifier attribute can be used to reference the hierarchy in [grouping with rollup](#sec_Groupingwithrollupandall).

### [Recursive Hierarchy](#sec_RecursiveHierarchy)

A *recursive hierarchy* organizes the values of a single groupable property as nodes of a tree structure. This structure does not need to be as uniform as a leveled hierarchy. It is described by a complex term RecursiveHierarchy with the properties:

* The NodeProperty contains the path to the identifier of the node.
* The ParentNavigationProperty allows navigation to the entity representing the parent node.
* The optional ChildrenNavigationProperty allows navigation to the entities representing the children nodes.
* The optional DistanceFromRootProperty contains the path to a property that contains the number of edges between the node and the root node.
* The optional IsLeafProperty contains the path to a Boolean property that indicates whether the node is a leaf of the hierarchy.

The term RecursiveHierarchy can only be applied to entity types, and the applying Annotation element MUST specify the Qualifier attribute. The value of the Qualifier attribute can be used to reference the hierarchy in [Hierarchy Functions](#sec_HierarchyFilterFunctions).

#### [Hierarchy Functions](#sec_HierarchyFilterFunctions)

For testing the position of a given entity instance in a recursive hierarchy annotated to the entity’s type, the Aggregation vocabulary defines functions that can be applied to any entity:

* isroot returns true if and only if the value of the node property of the specified hierarchy is the root of the hierarchy,
* isdescendant returns true if and only if the value of the node property of the specified hierarchy is a descendant of the given parent node with a distance of less than or equal to the optionally specified maximum distance or, if selected, optionally the parent node itself,
* isancestor returns true if and only if the value of the node property of the specified hierarchy is an ancestor of the given child node with a distance of less than or equal to the optionally specified maximum distance or, if selected, optionally the child node itself,
* issibling returns true if and only if the value of the node property of the specified hierarchy has the same parent node as the specified node,
* isleaf returns true if and only if the value of the node property of the specified hierarchy has no descendants.

### [Hierarchy Examples](#sec_HierarchyExamples)

Example 44: leveled hierarchies for products and time, and a recursive hierarchy for the sales organizations

<edmx:Edmx xmlns:edmx="<http://docs.oasis-open.org/odata/ns/edmx>"
 Version="4.0">
 <edmx:Reference Uri="http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/cs01/vocabularies/Org.OData.Aggregation.V1.xml">
 <edmx:Include Alias="Aggregation"
 Namespace="Org.OData.Aggregation.V1" />
 </edmx:Reference>

 <edmx:DataServices>

 <Schema xmlns="<http://docs.oasis-open.org/odata/ns/edm>"
 Alias="SalesModel" Namespace="org.example.odata.salesservice">
 <Annotations Target="SalesModel.Product">
 <Annotation Term="Aggregation.LeveledHierarchy"
 Qualifier="ProductHierarchy">
 <Collection>
 <PropertyPath>Category/Name</PropertyPath>
 <PropertyPath>Name</PropertyPath>
 </Collection>
 </Annotation>
 </Annotations>

 <Annotations Target="SalesModel.Time">
 <Annotation Term="Aggregation.LeveledHierarchy"
 Qualifier="TimeHierarchy">
 <Collection>
 <PropertyPath>Year</PropertyPath>
 <PropertyPath>Quarter</PropertyPath>
 <PropertyPath>Month</PropertyPath>
 </Collection>
 </Annotation>
 </Annotations>

 <Annotations Target="SalesModel.SalesOrganization">
 <Annotation Term="Aggregation.RecursiveHierarchy"
 Qualifier="SalesOrgHierarchy">
 <Record>
 <PropertyValue Property="NodeProperty"
 PropertyPath="ID" />
 <PropertyValue Property="ParentNavigationProperty"

 PropertyPath="Superordinate" />
 </Record>
 </Annotation>
 </Annotations>
 </Schema>
 </edmx:DataServices>
</edmx:Edmx>

The recursive hierarchy SalesOrgHierarchy can be used in functions with the $filter system query option.

Example 45: requesting all organizations below EMEA

GET ~/SalesOrganizations?
 $filter=
 $it/Aggregation.isdescendant(Hierarchy='SalesOrgHierarchy',Node='EMEA')

results in

{

 "@odata.context": "$metadata#SalesOrganizations",

 "value": [
 { "ID": "EMEA Central", "Name": "EMEA Central" },
 { "ID": "Sales Netherland", "Name": "Sales Netherland" },
 { "ID": "Sales Germany", "Name": "Sales Germany" },
 { "ID": "EMEA South", "Name": "EMEA South" },
 ...
 { "ID": "EMEA North", "Name": "EMEA North" },
 ...
 ]

}

Example 46: requesting just those organizations directly below EMEA

GET SalesOrganizations?$filter=
 $it/Aggregation.isdescendant(Hierarchy='SalesOrgHierarchy',
 Node='EMEA',MaxDistance=1)

results in

{

 "@odata.context": "$metadata#SalesOrganizations",

 "value": [
 { "ID": "EMEA Central", "Name": "EMEA Central" },
 { "ID": "EMEA South", "Name": "EMEA South" },
 { "ID": "EMEA North", "Name": "EMEA North" },
 ...
 ]

}

Example 47: just the lowest-level organizations

GET SalesOrganizations?$filter=
 $it/Aggregation.isleaf(Hierarchy='SalesOrgHierarchy')

results in

{

 "@odata.context": "$metadata#SalesOrganizations",

"value": [
 { "ID": "Sales Office London", "Name": "Sales Office London" },
 { "ID": "Sales Office New York", "Name": "Sales Office New York" },
 ...
 ]

}

Example 48: the lowest-level organizations including their superordinate's ID

GET SalesOrganizations?$filter=
 $it/Aggregation.isleaf(Hierarchy='SalesOrgHierarchy')
 &$expand=Superordinate($select=ID)

results in

{

 "@odata.context": "$metadata#SalesOrganizations(\*,Superordinate(ID))",

 "value": [
 { "ID": "Sales Office London", "Name": "Sales Office London",
 "Superordinate: { "ID": "EMEA United Kingdom" } },
 { "ID": "Sales Office New York", "Name": "Sales Office New York",
 "Superordinate: { "ID": "US East" } },
 ...
 ]

}

Example 49: retrieving the sales IDs involving sales organizations from EMEA can be requested by

GET Sales?$select=ID&$filter=

 SalesOrganization/Aggregation.isdescendant(Hierarchy='SalesOrgHierarchy',

 Node='EMEA')

results in

{

 "@odata.context": "$metadata#Sales(ID)",

 "value": [
 { "ID": 6 },
 { "ID": 7 },
 { "ID": 8 }
 ]

}

## [Actions and Functions on Aggregated Entities](#sec_ActionsandFunctionsonAggregatedEntit)

Bound actions and functions may or may not be applicable to aggregated entities. By default such bindings are not applicable to aggregated entities. Actions or functions annotated with the term AvailableOnAggregates are applicable to (a subset of the) aggregated entities under specific conditions:

* The RequiredProperties collection lists all properties that must be available in the aggregated entities; otherwise, the annotated function or action will be inapplicable.

Example 50: assume the product is an implicit input for a function bindable to Sales, then aggregating away the product makes this function inapplicable.

Calculating a set of aggregated entities and invoking an action on them cannot be accomplished with a single request, because the action URL cannot be constructed by the client. It is also impossible to construct a URL that calculates a single aggregated entity and applies a function or action on it. Consequently, applicable bound actions or functions on a single aggregated entity, or bound actions on a collection of aggregated entities MUST be advertised in the response to make them available to clients. A client is then able to request the aggregated entities in a first request and invoke the action or function in a follow-up request using the advertised target URL.

Example 51: full representation of an action applicable to a collection of aggregated entities, and an action that is applicable to one of the entities in the collection. The string <properties in $apply> is a stand-in for the list of properties describing the shape of the result set

{

"@odata.context": "$metadata#Sales(<properties in $apply>)",

"@odata.readLink": "http://.../aggregated-stuff2143248437259843",

"#Model.ColAction": {

 "title": "Do something on this collection",

 "target": "http://.../aggregated-stuff2143248437259843/Model.ColAction"

 },

"value": [

 {

 "@odata.id": "aggregated-stuff2143248437259843-1",

 "#Model.SingleAction": {

 "title": "Do something on this entity",

 "target":

 "http://.../aggregated-stuff2143248437259843-1/Model.SingleAction"

 },

 ...

 },

 ...

]

}

Services advertising the availability of functions or actions via the term AvailableOnAggregates MUST provide read links or edit links for aggregated entities, see section 4.

# [Examples](#sec_Examples)

The following examples show some common aggregation-related questions that can be answered by combining the transformations defined in chapter 3.

## [Requesting Distinct Values](#sec_RequestingDistinctValues)

Grouping without specifying a set transformation returns the distinct combination of the grouping properties.

Example 52:

GET ~/Customers?$apply=groupby((Name))

results in

{

 "@odata.context": "$metadata#Customers(Name)",

 "value": [
 { "Name": "Luc" },
 { "Name": "Joe" },
 { "Name": "Sue" }

 ]

}

Note that "Sue" appears only once although the customer base contains two different Sues.

Aggregation is also possible across related entities.

Example 53: customers that bought something

GET ~/Sales?$apply=groupby((Customer/Name))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Name))",

 "value": [
 { "Customer": { "Name": "Joe" } },
 { "Customer": { "Name": "Sue" } }

 ]

}

The result has the same structure as a standard OData request that expands the navigation properties and selects the data properties specified in groupby and aggregate.

GET ~/Sales?$expand=Customer($select=Name)

Note that "Luc" does not appear in the aggregated result as he hasn’t bought anything and therefore there are no sales entities that refer/navigate to Luc.

However, even though both Sues bought products, only one "Sue" appears in the aggregate result. Including properties that guarantee the right level of uniqueness in the grouping can repair that.

Example 54:

GET ~/Sales?$apply=groupby((Customer/Name,Customer/ID))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Name,ID))",

 "value": [
 { "Customer": { "Name": "Joe", "ID": "C1" } },
 { "Customer": { "Name": "Sue", "ID": "C2" } },

 { "Customer": { "Name": "Sue", "ID": "C3" } }

 ]

}

This could also have been formulated as

GET ~/Sales?$apply=groupby((Customer))

 &$expand=Customer($select=Name,ID)

Example 54: Grouping by navigation property Customer is treated as groupby((Customer/$ref))

GET ~/Sales?$apply=groupby((Customer))

results in

{

 "@odata.context": "$metadata#Sales(Customer)",

 "value": [
 { "Customer": { "@odata.id": "Customers('C1')" } },
 { "Customer": { "@odata.id": "Customers('C2')" } },

 { "Customer": { "@odata.id": "Customers('C3')" } }

 ]

}

Note: the typical representation of a deferred navigation property is a URL “relative” to the source entity, e.g. ~/Sales(1)/Customer. This has the benefit that this URL doesn’t change if the sales entity would be associated to a different customer. For aggregated entities this would actually be a drawback, so the representation MUST be the canonical URL of the target entity, i.e. ~/Customers('C1') for the first entity in the above result.

Example 55: the first question in the motivating example in section 2.4, which customers bought which products, can now be expressed as

GET ~Sales?$apply=groupby((Customer/Name,Customer/ID,Product/Name))

and results in

{

 "@odata.context": "$metadata#Sales(Customer(Name,ID),Product(Name))",

 "value": [
 { "Customer": { "Name": "Joe", "ID": "C1" },

 "Product": { "Name": "Coffee"} },
 { "Customer": { "Name": "Joe", "ID": "C1" },

 "Product": { "Name": "Paper" } },
 { "Customer": { "Name": "Joe", "ID": "C1" },

 "Product: { "Name: "Sugar" } },
 { "Customer": { "Name": "Sue", "ID": "C2" },

 "Product: { "Name": "Coffee"} },

 { "Customer": { "Name": "Sue", "ID": "C2" },

 "Product": { "Name": "Paper" } },

 { "Customer": { "Name": "Sue", "ID": "C3" },

 "Product": { "Name": "Paper" } },

 { "Customer": { "Name": "Sue", "ID": "C3" },

 "Product": { "Name": "Sugar" } }

 ]

}

## [Standard Aggregation Methods](#sec_StandardAggregationMethods)

The client may specify one of the predefined aggregation methods [min](#sec_StandardAggregationMethodmin), [max](#sec_StandardAggregationMethodmax), [sum](#sec_StandardAggregationMethodsum), [average](#sec_StandardAggregationMethodaverage), and [countdistinct](#sec_StandardAggregationMethodcountdistin), or a [custom aggregation method](#sec_CustomAggregationMethods), to aggregate an [aggregatable property](#sec_AggregatableProperties). Expressions defining an aggregate method MUST specify an [alias](#sec_Keywordas). The aggregated values are returned in a dynamic property whose name is determined by the alias.

Example 56:

GET ~/Products?$apply=groupby((Name),
 aggregate(Sales/Amount with sum as Total))

results in

{

 "@odata.context": "$metadata#Products(Name,Sales(Total))",

 "value": [
 { "Name": "Coffee", "Total": 12, ... },
 { "Name": "Paper", "Total": 8, ... },
 { "Name": "Pencil", "Total": null },
 { "Name": "Sugar", "Total": 4, ... }
 ]

}

Note that the base set of the request is Products, so there is a result item for product Pencil even though there are no sales item. As aggregate returns exactly one result item even if there are no items to be aggregated, the Sales navigation property’s value is an array with one element representing the sum over no input values, which is null.

Example 57: careful observers will notice that the above amounts have been aggregated across currencies, which is semantically wrong. Yet it is the correct response to the question asked, so be careful what you ask for. The semantically meaningful question

GET ~/Products?$apply=expand(Sales,groupby((Currency/Code),

 aggregate(Amount with sum as Total)))

results in

{

 "@odata.context": "$metadata#Products(Name,Sales(Total,Currency(Code)))",

 "value": [
 { "Name": "Coffee",

 "Sales": [ { "Total": 12, "Currency": { "Code": "USD" }, ... } ] },
 { "Name: "Paper",

 "Sales": [ { "Total": 3, "Currency": { "Code": "EUR" }, ... },
 { "Total": 5, "Currency": { "Code": "USD" }, ... } ] },
 { "Name: "Pencil", "Sales": [] },
 { "Name: "Sugar",
 "Sales": [ { "Total": 2, "Currency": { "Code": "EUR" }, ... },
 { "Total": 2, "Currency": { "Code": "USD" }, ... } ] }
 ]

}

Alternatively, join could be applied to yield a flat structure:

GET ~/Products?$apply=join(Sales,groupby((Currency/Code),

 aggregate(Amount with sum as Total)))

{

 "@odata.context": "$metadata#Products",

 "value": [

 { "Name": "Sugar", "Sales": { Currency: { Code: USD }, Total: 123 },

 { "Name": "Sugar", "Sales": { Currency: { Code: EUR }, Total: 456 },

… (to be completed)

 ]

}

Example 58:

GET ~/Customers?$apply=groupby((Country,Sales/Product/Name))

returns the different products sold per country:

{

 "@odata.context": "$metadata#Customers(Country,Sales(Product(Name)))",

 "value": [
 { "Country": "Netherlands",

 "Sales": [ { "Product": { "Name": "Paper" } },
 { "Product": { "Name": "Sugar" } } ] },
 { "Country": "USA",

 "Sales": [ { "Product": { "Name": "Coffee" } },
 { "Product": { "Name": "Paper" } },
 { "Product": { "Name": "Sugar" } } ] },

 { "Country": "France", "Sales": [ ] }

 ]

}

Example 59:

GET ~/Sales?$apply=groupby((Customer/Country),

 aggregate(Amount with average as AverageAmount))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country),AverageAmount)",

 "value": [
 { "Customer": { "Country": "Netherlands" }, "AverageAmount": 1.6666667 },
 { "Customer": { "Country": "USA" }, "AverageAmount": 3.8 }
 ]

}

If the example model would contain a list of hobbies per customer, with Hobbies a collection of strings, the number of different hobbies across the customer base could be requested. A navigation property followed by a /$count segment is a valid expression in the context that declares the navigation property, so the result property is placed in the same context as the navigation property.

Example 60:

GET ~/Products?$apply=groupby((Name),

 aggregate(Sales/$count with sum as SalesCount))

results in

{

 "@odata.context": "$metadata#Products(Name,SalesCount)",

 "value": [
 { "Name": "Coffee", "SalesCount": 2, ... },
 { "Name": "Paper", "SalesCount": 4, ... },
 { "Name": "Pencil", "SalesCount": 0, ... },
 { "Name": "Sugar", "SalesCount": 2, ... }
 ]

}

Note that this differs from the placement of an aggregated property in a related entity: the aggregated property has the same navigation path as the original value.

Example 61: the result properties for Sales/$count and Sales(Amount…) are placed differently

GET ~/Products?$apply=groupby((Name),

 aggregate(Sales/$count with sum as SalesCount,

 Sales(Amount with sum as TotalAmount)))

results in

{

 "@odata.context": "$metadata#Products(Name,SalesCount,Sales(TotalAmount))",

 "value": [
 { "Name": "Coffee", "SalesCount": 2, ...,

 "Sales": [ { "TotalAmount": 12, ... } ] },
 { "Name": "Paper", "SalesCount": 4, ...,
 "Sales": [ { "TotalAmount": 8, ... } ] },
 { "Name": "Pencil", "SalesCount": 0, ...,
 "Sales": [ { "TotalAmount": null } ] },
 { "Name": "Sugar", "SalesCount": 2, ...,

 "Sales": [ { "TotalAmount": 4, ... } ] }
 ]

}

To place the number of instances in a group next to other aggregated values, the aggregate expression [$count](#sec_VirtualPropertycount) can be used:

Example 62: the result properties for Sales/$count and Sales/Amount are placed differently

GET ~/Products?$apply=groupby((Name),aggregate(Sales($count as SalesCount),

 Sales(Amount with sum as TotalAmount)))

results in

{

 "@odata.context": "$metadata#Products(Name,Sales(SalesCount,TotalAmount))",

 "value": [
 { "Name": "Coffee",

 "Sales": [ { "SalesCount": 2, "TotalAmount": 12, ... } ] },
 { "Name": "Paper",
 "Sales": [ { "SalesCount": 4, "TotalAmount": 8, ... } ] },
 { "Name": "Pencil",
 "Sales": [ { "SalesCount": 0, "TotalAmount": null, ... } ] },
 { "Name": "Sugar",

 "Sales": [ { "SalesCount": 2, "TotalAmount": 4, ... } ] }
 ]

}

## [Requesting Custom Aggregates](#sec_RequestingCustomAggregates)

Custom aggregates are defined through the [CustomAggregate](#sec_CustomAggregates) annotation. They can be associated with either an entity type or an entity container.

A custom aggregate can be used by specifying the name of the custom aggregate in the [aggregate](#sec_Transformationaggregate) clause.

Example 63:

GET ~/Sales?$apply=groupby((Customer/Country),
 aggregate(Amount with sum as Actual,Forecast))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country),Actual,Forecast))",

"value": [
 { "Customer": { "Country": "Netherlands" },

 "Actual": 5, "Forecast": 4, ... },
 { "Customer": { "Country": "USA" },

 "Actual": 19, "Forecast": 21, ... }
 ]

}

The introduced dynamic properties MUST always be in the same set as the original property.

Example 64:

GET ~/Products?$apply=groupby((Name),
 aggregate(Sales(Amount with sum as Actual),
 Sales/Forecast))

results in

{

 "@odata.context": "$metadata#Products(Name,Sales(Actual,Forecast))",

 "value": [
 { "Name": "Coffee", "Sales": [ { "Actual": 12, "Forecast": 6, ... } ] },
 { "Name": "Paper", "Sales": [ { "Actual": 8, "Forecast": 2, ... } ] },
 { "Name": "Pencil", "Sales": [] },
 { "Name": "Sugar", "Sales": [ { "Actual": 4, "Forecast": 7, ... } ] }
 ]

}

When associated with an entity type a custom aggregate MAY have the same name as a property of the entity with the same type as the type returned by the custom aggregate. This is typically done when the aggregate is used as a default aggregate for that property.

Example 65 A custom aggregate can be defined with the same name as a property of the same type in order to define a default aggregate for that property.

GET ~/Sales?$apply=groupby((Customer/Country),aggregate(Amount))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country),Amount)",

 "value": [
 { "Customer": { "Country": "Netherlands" }, "Amount": 5, ... },
 { "Customer": { "Country": "USA" }, "Amount": 19, ... }
 ]
}

## [Aliasing](#sec_Aliasing)

A property can be aggregated in multiple ways, each with a different alias.

Example 66:

GET ~/Sales?$apply=groupby((Customer/Country),
 aggregate(Amount with sum as Total,
 Amount with average as AvgAmt))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country),Total,AvgAmt))",

"value": [
 { "Customer": { "Country": "Netherlands" },

 "Total": 5, "AvgAmt": 1.6666667, ... },
 { "Customer": { "Country": "USA" },

 "Total": 19, "AvgAmt": 3.8, ... }
 ]

}

The introduced dynamic properties MUST always be in the same set as the original property.

Example 67:

GET ~/Products?$apply=groupby((Name),
 aggregate(Sales(Amount with sum as Total),
 Sales(Amount with average as AvgAmt)))

results in

{

 "@odata.context": "$metadata#Products(Name,Sales(Total,AvgAmt))",

 "value": [
 { "Name": "Coffee", "Sales": [ { "Total": 12, "AvgAmt": 6, ... } ] },
 { "Name": "Paper", "Sales": [ { "Total": 8, "AvgAmt": 2, ... } ] },
 { "Name": "Pencil", "Sales": [ { "Total": null, "AvgAmt": null } ] },
 { "Name": "Sugar", "Sales": [ { "Total": 4, "AvgAmt": 2, ... } ] }
 ]

}

There is no hard distinction between groupable and aggregatable properties: the same property can be aggregated and used to group the aggregated results.

Example 68:

GET ~/Sales?$apply=groupby((Amount),aggregate(Amount with sum as Total))

will return all distinct amounts appearing in sales orders and how much money was made with deals of this amount

{

 "@odata.context": "$metadata#Sales(Amount,Total)",

 "value": [
 { "Amount": 1, "Total": 2, ... },
 { "Amount": 2, "Total": 6, ... },
 { "Amount": 4, "Total": 8, ... },
 { "Amount": 8, "Total": 8, ... }
 ]

}

## [Combining Transformations per Group](#sec_CombiningTransformationsperGroup)

Dynamic property names may be reused in different transformation sequences passed to concat.

Example 69: to get the best-selling product per country with sub-totals for every country, the partial results of a transformation sequence and a groupby transformation are concatenated:

GET ~/Sales?$apply=concat(

 groupby((Customer/Country,Product/Name,Currency/Code),
 aggregate(Amount with sum as Total))

 /groupby((Customer/Country,Currency/Code),

 topcount(1,Total)),

 groupby((Customer/Country,Currency/Code),

 aggregate(Amount with sum as Total)))

results in

{

 "@odata.context":
 "$metadata#Sales(Customer(Country),Product(Name),Total,Currency(Code))",

 "value": [
 { "Customer": { "Country": "USA" },
 "Product": { "Name": "Coffee" },
 "Total": 12, "Currency": { "Code": "USD" }, ...
 },
 { "Customer": { "Country": "Netherlands" },

 "Product": { "Name": "Paper" },
 "Total": 3, "Currency": { "Code": "EUR" }, ...
 },
 { "Customer": { "Country": "USA" },
 "Total": 19, "Currency": { "Code": "USD" }, ...
 },
 { "Customer": { "Country": "Netherlands" },
 "Total": 5, "Currency": { "Code": "EUR" }, ...
 }
 ]

}

Example 70: transformation sequences are also useful inside groupby: To get the aggregated amount by only considering the top two sales amounts per product and county:

GET ~/Sales?$apply=groupby((Customer/Country,Product/Name,Currency/Code),

 topcount(2,Amount)/aggregate(Amount with sum as Total))

results in

{

 "@odata.context":
 "$metadata#Sales(Customer(Country),Product(Name),Total,Currency(Code))",

 "value": [
 { "Customer": { "Country": "Netherlands" },

 "Product": { "Name": "Paper" },

 "Total": 3, "Currency": { "Code": "EUR" }, ...

 },
 { "Customer": { "Country": "Netherlands" },

 "Product": { "Name": "Sugar" },

 "Total": 2, "Currency": { "Code": "EUR" }, ...

 },
 { "Customer": { "Country": "USA" },

 "Product": { "Name": "Sugar" },

 "Total": 2, "Currency": { "Code": "USD" }, ...

 },
 { "Customer": { "Country": "USA" },

 "Product": { "Name": "Coffee" },

 "Total": 12, "Currency": { "Code": "USD" }, ...

 },
 { "Customer": { "Country": "USA" },

 "Product": { "Name": "Paper" },

 "Total": 5, "Currency": { "Code": "USD" }, ...

 }
 ]

}

## [Model Functions as Set Transformations](#sec_ModelFunctionsasSetTransformations)

Example 71: as a variation of the example shown in the previous section, a query for returning the best-selling product per country and the total amount of the remaining products can be formulated with the help of a model function.

For this purpose, the model includes a definition of a TopCountAndBalance function that accepts the count for the top entities in the given input set not to be considered for the balance:

<edm:Function Name="TopCountAndBalance"

 ReturnType="Collection(Edm.EntityType)"

 IsBound="true">

 <edm:Parameter Name="EntityCollection"

 Type="Collection(Edm.EntityType)"/>

 <edm:Parameter Name="Count" Type="Edm.Int16"/>

 <edm:Parameter Name="Property" Type="Edm.String"/>

</edm:Function>

The function takes the name of a numeric property as a parameter, retains those entities that topcount also would retain, and replaces the remaining entities by a single aggregated entity, where only the numeric property has a defined value being the aggregated value over those remaining entities:

GET ~/Sales?$apply=groupby((Customer/Country,Product/Name),

 aggregate(Amount with sum as Total))

 /groupby((Customer/Country),

 Self.TopCountAndBalance(Count=1,Property='Total'))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country),Product(Name),Total)",

 "value": [
 { "Customer": { "Country": "Netherlands" },

 "Product": { "Name": "Paper" }, "Total": 3, ... },
 { "Customer": { "Country": "Netherlands" }, "Total": 2, ... },
 { "Customer": { "Country": "USA" },

 "Product": { "Name": "Coffee" }, "Total": 12, ... },
 { "Customer": { "Country": "USA" }, "Total": 7, ... }
 ]

}

Note that these two entities get their values for the Country property from the groupby transformation, which ensures that they contain all grouping properties with the correct values.

## [Controlling Aggregation per Rollup Level](#sec_ControllingAggregationperRollupLevel)

Consumers may specify a different aggregation method per level for every property passed to [rollup](#sec_Groupingwithrollupandall) as a hierarchy level below the root level.

Example 72: get the average of the overall amount by month per product.

Using a transformation sequence:

GET ~/Sales?$apply=groupby((Product/ID,Product/Name,Time/Month),
 aggregate(Amount with sum) as Total))
 /groupby((Product/ID,Product/Name),
 aggregate(Total with average as AverageAmount))

Using [from](#sec_Keywordfrom):

GET ~/Sales?$apply=groupby((Product/ID,Product/Name),
 aggregate(Amount with sum

 from Time/Month with average

 as MonthlyAverage))

Example 73: for an aggregate entity set listing the total sales amounts per customer and country, the rollup shall produce additional instances for the average total sales amount of customers per country and the average of that average (which is a bit boring because the example data doesn’t have two countries with the same currencyJ)

GET ~/Sales?$apply=groupby((rollup($all,Customer/Country,Customer/ID),
 Currency/Code),
 aggregate(Amount with sum
 from Customer/ID with average
 from Customer/Country with average

 as CustomerCountryAverage))

results in

{

 "@odata.context": "$metadata#Sales(Customer(Country,ID),CustomerCountryAverage,Currency(Code))",

 "value": [
 { "Customer": { "Country": "USA", "ID": "C1" },
 "CustomerCountryAverage": 7, "Currency": { "Code": "USD" }
 },
 { "Customer": { "Country": "USA", "ID": "C2" },
 "CustomerCountryAverage": 12, "Currency": { "Code": "USD" }
 },
 { "Customer": { "Country": "USA" },
 "CustomerCountryAverage": 9.5, "Currency": { "Code": "USD" }
 },
 { "Customer": { "Country": "Netherlands", "ID": "C3" },
 "CustomerCountryAverage": 5, "Currency": { "Code": "EUR" }
 },
 { "Customer": { "Country": "Netherlands" },
 "CustomerCountryAverage": 5, "Currency": { "Code": "EUR" }
 },
 { "CustomerCountryAverage": 9.5, "Currency": { "Code": "USD" }
 },
 { "CustomerCountryAverage": 5, "Currency": { "Code": "EUR" }
 }
 ]

}

## [Transformation Sequences](#sec_TransformationSequences)

Applying aggregation first covers the most prominent use cases. The slightly more sophisticated question "how much money is earned with small sales" requires filtering the base set before applying the aggregation. To enable this type of question several transformations can be specified in $apply in the order they are to be applied, separated by a forward slash.

Example 74:

GET ~/Sales?$apply=filter(Amount le 1)/aggregate(Amount with sum as Total)

means "filter first, then aggregate", and results in

{

 "@odata.context": "$metadata#Sales(Total)",

 "value": [
 { "Total": 2, ... }
 ]

}

Using filter within $apply does not preclude using it as a normal system query option.

Example 75:

GET ~/Sales?$apply=filter(Amount le 2)/groupby((Product/Name),
 aggregate(Amount with sum as Total))
 &$filter=Total ge 4

results in

{

 "@odata.context": "$metadata#Sales(Product(Name),Total)",

 "value": [
 { "Total": 4, "Product": { "Name": "Paper" }, ... },
 { "Total": 4, "Product": { "Name": "Sugar" }, ... }
 ]

}

Example 76: Revisiting the Example 14 for using the from keyword with the aggregate function, the request

GET ~/Sales?$apply=aggregate(Amount from Time with average as DailyAverage)

could be rewritten in a more procedural way using a transformation sequence returning the same result

GET ~/Sales?$apply=groupby((Time),aggregate(Amount with sum as Total))
 /aggregate(Total with average as DailyAverage)

For further examples, consider another data model containing entity sets for cities, countries and continents and the obvious associations between them.

Example 77: getting the population per country with

GET ~/Cities?$apply=groupby((Continent/Name,Country/Name),
 aggregate(Population with sum as TotalPopulation))

results in

{

 "@odata.context":

 "$metadata#Cities(Continent(Name),Country(Name),TotalPopulation)",

 "value": [
 { "Continent": { "Name": "Asia" },

 "Country": { "Name": "China" }, "TotalPopulation": 692.580.000, ... },
 { "Continent": { "Name": "Asia" },

 "Country": { "Name": "India" }, "TotalPopulation": 390.600.000, ... },
 ...
 ]

}

Example 78: all countries with megacities and their continents

GET ~/Cities?$apply=filter(Population ge 10000000)
 /groupby((Continent/Name,Country/Name),
 aggregate(Population with sum as TotalPopulation))

Example 79: all countries with tens of millions of city dwellers and the continents only for these countries

GET ~/Cities?$apply=groupby((Continent/Name,Country/Name),
 aggregate(Population with sum as CountryPopulation))
 /filter(CountryPopulation ge 10000000)
 /concat(identity,
 groupby((Continent/Name),
 aggregate(CountryPopulation with sum

 as TotalPopulation)))

– OR –

GET ~/Cities?$apply=groupby((Continent/Name,Country/Name),
 aggregate(Population with sum as CountryPopulation))
 /filter(CountryPopulation ge 10000000)
 /groupby((rollup(Continent/Name,Country/Name)),
 aggregate(CountryPopulation with sum

 as TotalPopulation))

Example 80: all countries with tens of millions of city dwellers and all continents with cities independent of their size

GET ~/Cities?$apply=groupby((Continent/Name,Country/Name),
 aggregate(Population with sum as CountryPopulation))
 /concat(filter(CountryPopulation ge 10000000),
 groupby((Continent/Name),
 aggregate(CountryPopulation with sum

 as TotalPopulation)))

Example 81: assuming the data model includes a sales order entity set with related sets for order items and customers, the base set as well as the related items can be filtered before aggregation

GET ~/SalesOrders?$apply=filter(Status eq 'incomplete')
 /expand(Items,filter(not Shipped))
 /groupby((Customer/Country),
 aggregate(Items/Amount with sum as ItemAmount))

# [Conformance](#sec_Conformance)

Conforming services MUST follow all rules of this specification for the set transformations and aggregation methods they support. They MUST implement all set transformations and aggregation methods they advertise via the annotation [ApplySupported](#sec_AggregationCapabilities).

Conforming clients MUST be prepared to consume a model that uses any or all of the constructs defined in this specification, including custom aggregation methods defined by the service, and MUST ignore any constructs not defined in this version of the specification.

1. [Acknowledgments](#sec_Acknowledgments)

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

1. [Revision History](#sec_RevisionHistory)

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Editor** | **Changes Made** |
| Working Draft 01 | 2012-11-12 | Ralf Handl | Translated contribution into OASIS format |
| Committee Specification Draft 01 | 2013-07-25 | Ralf HandlHubert HeijkersGerald KrauseMichael PizzoMartin Zurmuehl | Switched to pipe-and-filter-style query language based on composable set transformationsFleshed out examples and addressed numerous editorial and technical issues processed through the TCAdded Conformance section |
| Committee Specification Draft 02 | 2014-01-09 | Ralf HandlHubert HeijkersGerald KrauseMichael PizzoMartin Zurmuehl | Dynamic properties used all aggregated values. either via aliases or via custom aggregatesRefactored annotations |
| Committee Specification Draft 03 | 2015-07-16 | Ralf HandlHubert HeijkersGerald KrauseMichael PizzoMartin Zurmuehl | Added compute transformationMinor clean-up |
| Committee Specification Draft 03 |  | Ralf HandlHubert HeijkersGerald KrauseMichael PizzoMartin Zurmuehl |  |