OData Extension for Data Aggregation Version 4.0

Committee Specification Draft 01 / Public Review Draft 01

25 July 2013

Specification URIs

This version:

http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd01/odata-data-aggregation-ext-v4.0-csprd01.docx (Authoritative)
http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd01/odata-data-aggregation-ext-v4.0-csprd01.html
http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd01/odata-data-aggregation-ext-v4.0-csprd01.pdf

Previous version:

N/A

Latest version:

http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/odata-data-aggregation-ext-v4.0.docx (Authoritative)
http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/odata-data-aggregation-ext-v4.0.html
http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/odata-data-aggregation-ext-v4.0.pdf

Technical Committee:

OASIS Open Data Protocol (OData) TC

Chairs:

Barbara Hartel (barbara.hartel@sap.com), SAP AG
Ram Jeyaraman (Ram.Jeyaraman@microsoft.com), Microsoft

Editors:

Ralf Handl (ralf.handl@sap.com), SAP AG
Hubert Heijkers (hubert.heijkers@nl.ibm.com), IBM
Gerald Krause (gerald.krause@sap.com), SAP AG
Michael Pizzo (mikep@microsoft.com), Microsoft
Martin Zurmuehl (martin.zurmuehl@sap.com), SAP AG

Additional artifacts:

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

- OData Aggregation ABNF Construction Rules Version 4.0 and OData Aggregation ABNF Test Cases: http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd01/abnf/
- OData Aggregation Vocabulary: http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd01/vocabularies/

Related work:

This specification is related to:

- OData Version 4.0, a multi-part Work Product that includes:
  - OData Version 4.0 Part 1: Protocol. Latest version. http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part1-protocol.html
  - OData Version 4.0 Part 2: URL Conventions. Latest version. http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part2-url-conventions.html
  - OData Version 4.0 Part 3: Common Schema Definition Language (CSDL). Latest version. http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part3-csdl.html
  - ABNF components: OData ABNF Construction Rules Version 4.0 and OData ABNF Test Cases. 24 June 2013. http://docs.oasis-open.org/odata/odata/v4.0/csprd02/abnf/
  - Vocabulary components: OData Core Vocabulary and OData Measures Vocabular. 24 June 2013. http://docs.oasis-open.org/odata/odata/v4.0/csprd02/vocabularies/
- OData JSON Format Version 4.0. Latest version. http://docs.oasis-open.org/odata/odata-json-format/v4.0/odata-json-format-v4.0.html

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 document was last revised or approved by the OASIS Open Data Protocol (OData) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at http://www.oasis-open.org/committees/odata/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (http://www.oasis-open.org/committees/odata/ipr.php).

Citation format:

When referencing this specification the following citation format should be used:

[OData-Data-Agg-v4.0]

OData Extension for Data Aggregation Version 4.0. 25 July 2013. OASIS Committee Specification Draft 01 / Public Review Draft 01. http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd01/odata-data-aggregation-ext-v4.0-csprd01.html.

Notices

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 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.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see http://www.oasis-open.org/policies-guidelines/trademark for above guidance.

Table of Contents

1 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.

1.1 Terminology

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

1.2 Normative References

[OData-ABNF] *OData ABNF Construction Rules Version 4.0*.
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-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.

1.3 Non-Normative References

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

1.4 Typographical Conventions

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

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

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

Example 1: text describing an example uses this paragraph style

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

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

All other text is normative unless otherwise labeled.

2 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]**) 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 explicitly express the desired aggregation behavior for a particular query. They also allow consumers to formulate queries that refer to the annotations as shorthand.

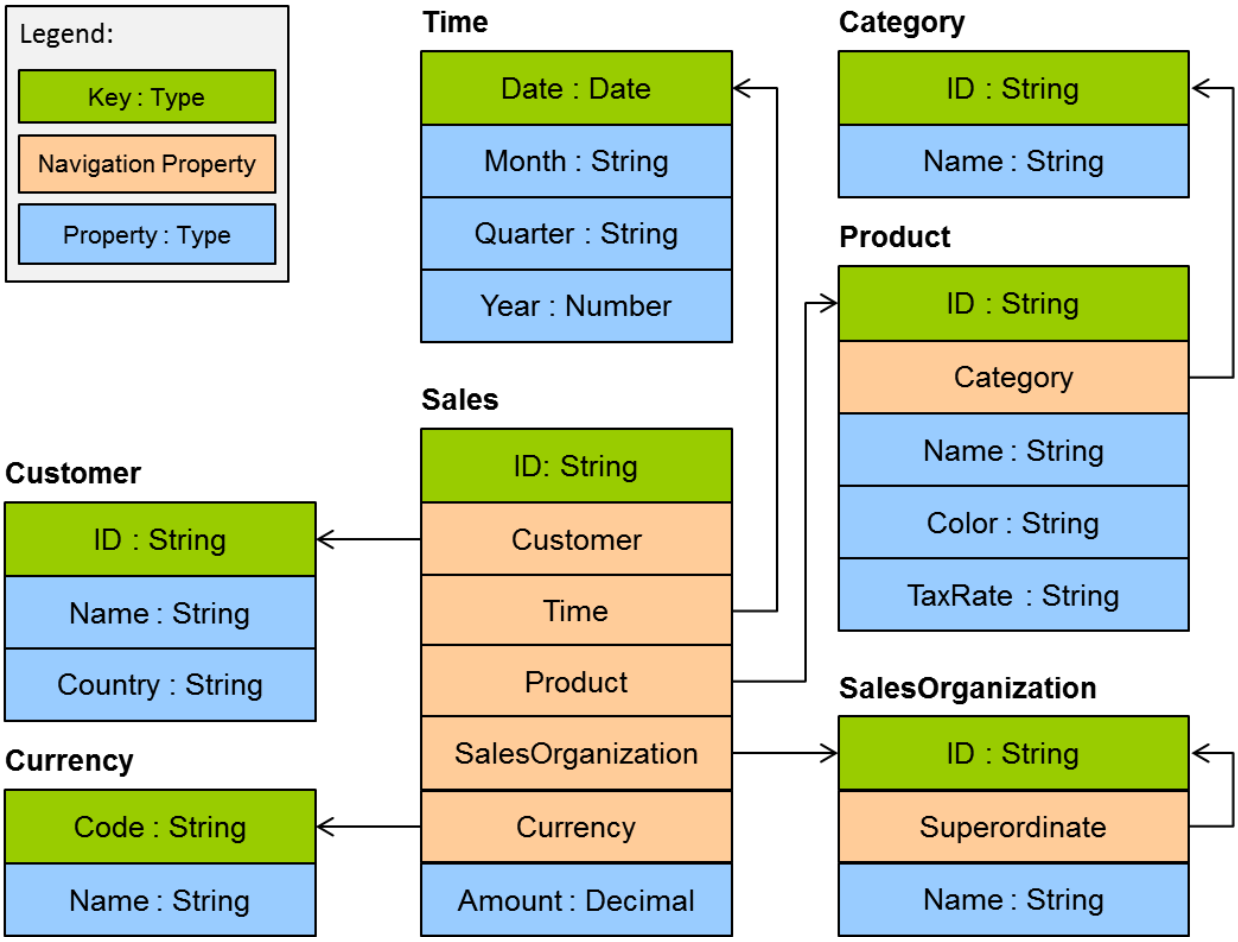
2.1 Definitions

This specification defines the following terms:

- *Aggregatable property* – a property for which the values can be aggregated.
- *Groupable property* – a property whose values can be used to group entities for aggregation.
- *Hierarchy* – an arrangement of groupable properties whose values are represented as being “above”, “below”, or “at the same level as” one another.

2.2 Example Data Model

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



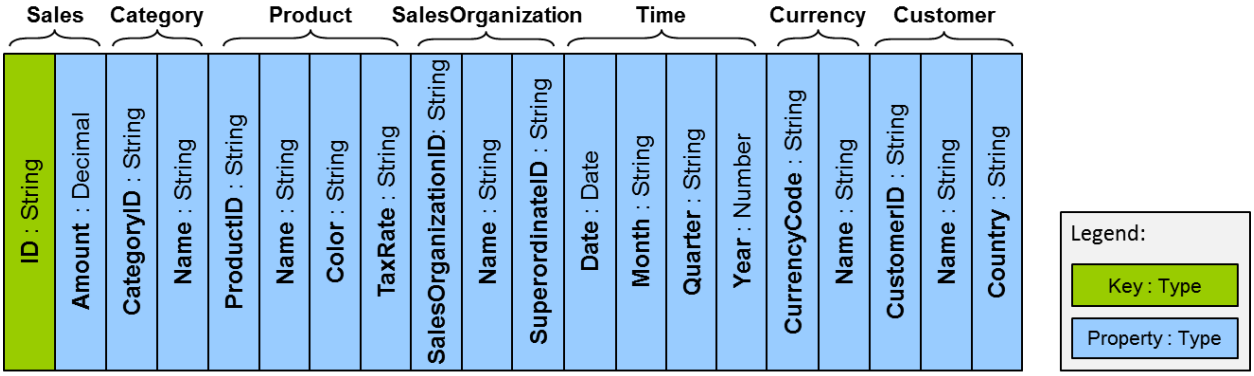
The Amount property in the Sales entity type is an aggregatable property, and the properties of the related entity types are groupable. These can be arranged in four hierarchies:

- *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*

The examples throughout the document, based on this diagram, assume that the default aggregation behavior of Amount has been set to summation. To further assist consumers of this service, the service metadata is accordingly annotated using the vocabulary defined in section 6.

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.

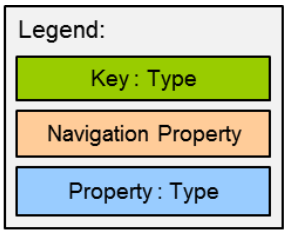
Query extensions and descriptive annotations can both be applied to normalized as well as partly or fully denormalized schemas.



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.

2.3 Example Data

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



Products

ID	Category	Name	Color	TaxRate
P1	PG1	Sugar	White	0.06
P2	PG1	Coffee	Brown	0.06
P3	PG2	Paper	White	0.14
P4	PG2	Pencil	Black	0.14

Categories

ID	Name
PG1	Food
PG2	Non-Food

Time

Date	Month	Quarter	Year
2012-01-01	2012-01	2012-1	2012
2012-04-01	2012-04	2012-2	2012
2012-04-10	2012-04	2012-2	2012
…			

SalesOrganizations

ID	Superordinate	Name
Sales		Corporate Sales
US	Sales	US
US West	US	US West
US East	US	US East
…		
EMEA	Sales	EMEA
EMEA Central	EMEA	EMEA Central
…		

Customers

ID	Name	Country
C1	Joe	USA
C2	Sue	USA
C3	Sue	Netherlands
C4	Luc	France

Currencies

Code	Name
USD	US Dollar
EUR	Euro

Sales

ID	Customer	Time	Product	Sales Organization	Currency	Amount
1	C1	2012-01-03	P3	US West	USD	1
2	C1	2012-04-10	P1	US West	USD	2
3	C1	2012-08-07	P2	US West	USD	4
4	C2	2012-01-03	P2	US East	USD	8
5	C2	2012-11-09	P3	US East	USD	4
6	C3	2012-04-01	P1	EMEA Central	EUR	2
7	C3	2012-08-06	P3	EMEA Central	EUR	1
8	C3	2012-11-22	P3	EMEA Central	EUR	2

2.4 Example Use Cases

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.

3 System Query Option `$apply`

Aggregation behavior 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, e.g. 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 entities, so the structure of the result fits into the data model of the service.
- does not necessarily preserve the number of result entries, as this will typically differ from the number of input entities.
- does not necessarily guarantee that all properties of the result entities have a well-defined value.

So the actual (or relevant) structure of each intermediary result will 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]**. The parameters of set transformations allow specifying how the result entities are constructed from the input entities.

The set transformations defined by this extension are

- `aggregate`
- `topcount`
- `topsum`
- `toppercent`
- `bottomcount`
- `bottomsum`
- `bottompercent`
- `identity`
- `concat`
- `groupby`
- `filter`
- `expand`

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

3.1 Transformation `aggregate`

The `aggregate` transformation takes zero or more parameters and returns a result set with a single entity, representing the aggregated value for all entities in the input set.

In its simplest form, it takes one or more paths to aggregatable properties as parameters. The resulting entity contains those properties (similar to a `$select` system query option), and their values represent the aggregated value of those properties across all entities of the input set, using the provider-defined default aggregation behavior. If paths are present, the corresponding navigation properties are implicitly expanded to make the properties part of the result representation.

Applying `aggregate` without parameters produces a set with one entity that does not have any properties.

Example 2:

```
GET ~/Sales?$apply=aggregate(Amount)
```

results in

```
{
  "odata.context": "$metadata#Sales(Amount)",
  "value": [
    { "odata.id": null, "Amount": 24 }
  ]
}
```

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

```
GET ~/Sales?$select=Amount
```

which would result in eight entities.

The `aggregate` transformation affects the structure of the result set: each parameter property corresponds to an item in a `$select` option. If the parameter property is preceded by a navigation path, the corresponding `$select` option would be nested in one `$expand` option for each navigation property in the navigation path.

3.1.1 Keyword `as`

When aggregating a simple property, an alias MAY be specified with the `as` keyword, followed by a SimpleIdentifier (see **[OData-CSDL, section 19.2]**).

The alias will introduce a dynamic property in the entity type of the aggregated. The introduced dynamic property MUST always be in the same entity type as the original property. The alias MUST NOT collide with names of declared properties or other aliases in that entity type.

Example 3:

```
GET ~/Sales?$apply=aggregate(Amount as TotalAmount)
```

results in

```
{
  "odata.context": "$metadata#Sales(TotalAmount)",
  "value": [
    { "odata.id": null, "TotalAmount": 24 }
  ]
}
```

When aggregating the result of an arithmetic expression, an alias MUST be specified with the `as` keyword. The expression MUST result in a simple value and MUST be an expression that could also be used in a `$filter` system query option on the input set.

Example 4:

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

results in

```
{
  "odata.context": "$metadata#Sales(Tax)",
  "value": [
    { "odata.id": null, "Tax": 2.08 }
  ]
}
```

If the 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. The expression within the

parentheses MUST be an expression that could also be used in a `$filter` system query option on the related entities identified by the navigation path. This syntax is intentionally similar to the syntax of `$expand` with nested query options.

Example 5:

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

results in

```
{
  "odata.context": "$metadata#Products(Sales(Tax))",
  "value": [
    { "odata.id": null, "Sales": [ { "Tax": 2.08 } ] }
  ]
}
```

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

```
GET ~/Products?$expand=Sales($select=Tax)
```

An alias affects the structure of the result set: each alias corresponds to an item in a `$select` option that is nested in an `$expand` option for each navigation property in the path of the aliased property.

3.1.2 Keyword `with`

To override the default aggregation behavior, a property path MAY be followed by the keyword `with` and followed by the name of the aggregation method to apply.

The predefined aggregation methods are `sum`, `min`, `max`, `average`, and `countdistinct`.

Providers MAY define non-standard aggregation methods using annotations, see section 6.1. These aggregation methods MUST use a namespace-qualified name (see **[OData-ABNF]**), i.e. contain at least one dot. Dot-less names are reserved for future versions of this specification.

3.1.2.1 Standard Aggregation Method `sum`

The standard aggregation method `sum` MAY be applied to a numeric aggregatable property. It returns the sum of the non-null values, and null if all values are null.

If an alias is introduced with the `as` keyword, the provider MUST choose a type for the result property that is capable of representing the aggregated values. This may require switching to a larger integer type, to `Edm.Decimal` with sufficient `Precision` and `Scale`, or to `Edm.Double`.

If no alias is introduced, the aggregated property will contain the result. If the aggregated property has type `Edm.Decimal` or one of the integer types and the result exceeds the limits of the integer part of the type, the producer MUST respond with `400 Bad Request` and a meaningful human-readable error message

Example 6:

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

results in

```
{
  "odata.context": "$metadata#Sales(TotalAmount)",
  "value": [
    { "odata.id": null, "TotalAmount": 24 }
  ]
}
```


3.1.2.2 Standard Aggregation Method `min`

The standard aggregation method `min` MAY be applied to an aggregatable property with a totally ordered domain. It returns the smallest of the non-null values, and null if all values are null.

If an alias is introduced with the `as` keyword, the result property will have the same type as the input property.

Example 7:

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

results in

```
{
  "odata.context": "$metadata#Sales(MinAmount)",
  "value": [
    { "odata.id": null, "MinAmount": 1 }
  ]
}
```

3.1.2.3 Standard Aggregation Method `max`

The standard aggregation method `max` MAY be applied to an aggregatable property with a totally ordered domain. It returns the largest of the non-null values, and null if all values are null.

If an alias is introduced with the `as` keyword, the result property will have the same type as the input property

Example 8:

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

results in

```
{
  "odata.context": "$metadata#Sales(MinAmount)",
  "value": [
    { "odata.id": null, "MaxAmount": 8 }
  ]
}
```

3.1.2.4 Standard Aggregation Method `average`

The standard aggregation method `average` MAY be applied to a numeric aggregatable property. It returns the sum of the non-null values divided by the count of the non-null values, and null if all values are null.

If an alias is introduced with the `as` keyword, the result property will be of type `Edm.Double`.

If no alias is introduced, the aggregated property will contain the result, and the result will be rounded according to the rounding rules defined in **[OData-URL]**.

Example 9:

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

results in

```
{
  "odata.context": "$metadata#Sales(AverageAmount)",
  "value": [
    { "odata.id": null, "AverageAmount": 3.0 }
  ]
}
```


3.1.2.5 Standard Aggregation Method `countdistinct`

The aggregation method `countdistinct` counts the distinct values of a property, omitting any null values. For navigation properties it counts the distinct entities (comparing the key property values) 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.

An alias MUST be introduced with the `as` keyword, and the result property MUST have type `Edm.Decimal` with `Scale="0"` and sufficient `Precision`.

Example 10:

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

results in

```
{
  "odata.context": "$metadata#Sales(DistinctProducts)",
  "value": [
    { "odata.id": null, "DistinctProducts": 3 }
  ]
}
```

3.1.3 Keyword `from`

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 aggregated value of some property 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, and optionally followed by a `with` clause. If the `with` clause is omitted, the provider-controlled default aggregation method is applied:

$aggregatableProperty$ [with $aggregationMethod$ as $alias$]
from $groupableProperty_1$ [with $aggregationMethod_1$]
…
from $groupableProperty_n$ [with $aggregationMethod_n$]

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

If the `from` keyword is present, first the default aggregation method (provider-controlled or explicitly specified for the aggregatable property) is used to aggregate away properties that don't have a special aggregation method specified and are not grouping properties.

Then consecutively properties not part of the result are aggregated away using the specific method in the order of the `from` clauses.

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

groupby((*groupableProperty*$_1$, …, *groupableProperty*$_n$),aggregate(*aggregatableProperty*))
/groupby((*groupableProperty*$_2$, …, *groupableProperty*$_n$),
aggregate(*aggregatableProperty* with *aggregationMethod*$_1$ as *aggregateableProperty*$_{tmp,1}$))
…
/groupby((*groupableProperty*$_n$),
aggregate(*aggregatableProperty*$_{tmp,n-2}$ with *aggregationMethod*$_{n-1}$ as *aggregationProperty*$_{tmp,n-1}$))

/aggregate(*aggregatableProperty*$_{tmp,n-1}$ with *aggregationMethod*$_n$ as *alias*))

The order of `from` clauses has to be compatible with hierarchies referenced from a leveled hierarchy annotation or specified as an unnamed hierarchy in groupby with rollup: 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.

Example 11:

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

is equivalent to

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

and results in the average sales volume per day

```
{
  "odata.context": "$metadata#Sales(DailyAverage)",
  "value": [
    { "odata.id": null, "DailyAverage": 3.428571428571429 }
  ]
}
```

3.1.4 Virtual Property `$count`

The value of the virtual property `$count` is the number of entities in a group. It MUST always specify an alias with the `as` keyword and MUST NOT specify an aggregation method with the `with` keyword.

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

Example 12:

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

results in

```
{
  "odata.context": "$metadata#Sales(SalesCount)",
  "value": [
    { "odata.id": null, "SalesCount": 8 }
  ]
}
```

3.2 Transformation `topcount`

The `topcount` transformation takes two parameters.

The first parameter specifies the number of entities to return in the transformed set. It MUST be an expression that can be evaluated on the set level and MUST result in a positive integer.

The second parameter specifies the value by which the entities are compared for determining the result set. It MUST be an expression that can be evaluated on entities of the input set and MUST result in a primitive numeric value.

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

In case the value of the second expression is ambiguous, the ordering of the input set is additionally taken into account.

Example 13:

```
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.

3.3 Transformation `topsum`

The `topsum` transformation takes two parameters.

The first parameter indirectly specifies the number of entities to return in the transformed set. It MUST be an expression that can be evaluated on the set level and MUST result in a number.

The second parameter specifies the value by which the entities are compared for determining the result set. It MUST be an expression that can be evaluated on entities of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of entities 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 entities in the input set.

In case the value of the second expression is ambiguous, the ordering of the input set is additionally taken into account.

Example 14:

```
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.

3.4 Transformation `toppercent`

The `toppercent` transformation takes two parameters.

The first parameter specifies the number of entities to return in the transformed set. It MUST be an expression 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 entities are compared for determining the result set. It MUST be an expression that can be evaluated on entities of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of entities 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 entities in the input set specified by the first parameter. It does not change the order of the entities in the input set.

In case the value of the second expression is ambiguous, the ordering of the input set is additionally taken into account.

Example 15:

```
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.

3.5 Transformation `bottomcount`

The `bottomcount` transformation takes two parameters.

The first parameter specifies the number of entities to return in the transformed set. It MUST be an expression that can be evaluated on the set level and MUST result in a positive integer.

The second parameter specifies the value by which the entities are compared for determining the result set. It MUST be an expression that can be evaluated on entities of the input set and MUST result in a primitive numeric value.

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

In case the value of the second expression is ambiguous, the ordering of the input set is additionally taken into account.

Example 16:

```
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.

3.6 Transformation `bottomsum`

The `bottomsum` transformation takes two parameters.

The first parameter indirectly specifies the number of entities to return in the transformed set. It MUST be an expression that can be evaluated on the set level and MUST result in a number.

The second parameter specifies the value by which the entities are compared for determining the result set. It MUST be an expression that can be evaluated on entities of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of entities 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 entities in the input set.

In case the value of the second expression is ambiguous, the ordering of the input set is additionally taken into account.

Example 17:

```
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.

3.7 Transformation `bottompercent`

The `bottompercent` transformation takes two parameters.

The first parameter indirectly specifies the number of entities to return in the transformed set. It MUST be an expression 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 entities are compared for determining the result set. It MUST be an expression that can be evaluated on entities of the input set and MUST result in a primitive numeric value.

The transformation returns the minimum set of entities 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 entities in the input set specified by the first parameter. It does not change the order of the entities in the input set.

In case the value of the second expression is ambiguous, the ordering of the input set is additionally taken into account.

Example 18:

```
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.

3.8 Transformation `identity`

The `identity` transformation returns its input set.

Example 19:

```
GET ~/Sales?$apply=identity
```


3.9 Transformation `concat`

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

It applies each parameter to the input set and concatenates the intermediate result sets in the order of the parameters into the result set, keeping the ordering of the individual result sets.

Example 20:

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

results in

```
{
  "odata.context": "$metadata#Sales",
  "value": [
    { "ID": 3, "Amount": 4, ... },
    { "ID": 4, "Amount": 8, ... },
    { "ID": 1, "Amount": 1, ... },
    { "ID": 7, "Amount": 1, ... }
  ]
}
```

The result set of `concat` has a hybrid form consisting of the structure imposed by the two transformation sequences.

3.10 Transformation `groupby`

The `groupby` transformation groups the entities of the input set into subsets, transforms each subset, and then concatenates the transformed subsets. It takes two parameters, the first specifying how the input set is split into subsets, the second how each subset is transformed. This transformation may remove properties from the input set or add new properties to it (see `aggregate`) as well as the number of entities it contains.

3.10.1 Simple Grouping

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. 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 second argument 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.

The `groupby` transformation:

- Splits the initial set into subsets where all entities in a subset have the same values for the grouping properties specified in the first parameter,
- Applies the set transformations to each subset, resulting in a new set of potentially different cardinality,
- Ensures that the entities in the result set contain all grouping properties with the correct values for the group,
- Concatenates the intermediate result sets into one result set.

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 entities indirectly by another property related to it in some way.

Example 21:

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

results in

```
{
  "odata.context": "$metadata#Sales(Customer(Country),Product(Name),Amount)",
  "value": [
    { "odata.id": null, "Customer": { "Country" : "Netherlands" },
      "Product": { "Name": "Paper" }, "Amount": 3 },
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Product": { "Name": "Sugar" }, "Amount": 2},
    { "odata.id": null, "Customer": { "Country: "USA" },
      "Product": { "Name": "Coffee" }, "Amount": 12},
    { "odata.id": null, "Customer": { "Country: "USA" },
      "Product": { "Name": "Paper" },"Amount": 5},
    { "odata.id": null, "Customer": { "Country: "USA" },
      "Product": { "Name": "Sugar" }, "Amount": 2}
  ]
}
```

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

```
GET ~/Sales?$expand=Customer($select=Country),Product($select=Name)
           &$select=Amount
```

Combined with `aggregate` without parameters `groupby` produces the distinct value combinations of the grouping properties. As this is an important special case, the second parameter of `groupby` can be omitted to request distinct value combinations.

Example 22: Both requests are equivalent

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

and result in

```
{
  "odata.context": "$metadata#Sales(Product(Name),Amount)",
  "value": [
    { "odata.id": null, "Product": { "Name": "Coffee" }, "Amount": 4 },
    { "odata.id": null, "Product": { "Name": "Coffee" }, "Amount": 8 },
    { "odata.id": null, "Product": { "Name": "Paper"  }, "Amount": 1 },
    { "odata.id": null, "Product": { "Name": "Paper"  }, "Amount": 2 },
    { "odata.id": null, "Product": { "Name": "Paper"  }, "Amount": 4 },
    { "odata.id": null, "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: each grouping property corresponds to an item in a `$select` option that is nested in an `$expand` option for each navigation property in the path of the grouping property. The set transformations used the second parameter of `groupby` furthermore affect the structure as described for every transformation.

3.10.2 Grouping with `rollup` and `$all`

The `rollup` grouping operator allows query option is used to 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` prefixed with the navigation path leading to the annotated entity type. This named hierarchy is used for grouping entities.

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 entities.

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 p_i 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`,…,` p_{n-1}`,` p_n`),` R`),` T`)` is equivalent to `concat(groupby((`p_1`,…,` p_{n-1}`,` p_n`,` R`),` T`),groupby((rollup(`p_1`,…,` p_{n-1}`),` R`),` T`))`
- `groupby((rollup(`p_1`,` p_2`),` R`),` T`)` is equivalent to `concat(groupby((`p_1`,` p_2`,` R`),` T`), groupby((`p_1`,` 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 23: rolling up two hierarchies, the first with two levels, the second with three levels:

```
(rollup(p1,1, p1,2),rollup(p2,1, p2,2, p2,3))
```

will result in the six groupings

```
(p1,1, p1,2,          p2,1, p2,2, p2,3)
(p1,1, p1,2,          p2,1, p2,2)
(p1,1, p1,2,          p2,1)
(p1,1,                p2,1, p2,2, p2,3)
(p1,1,                p2,1, p2,2)
(p1,1,                p2,1)
```

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 entities within detail entities is up to the service if no `$orderby` is given, otherwise at the position determined by `$orderby`.

Example 24: 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))
```

results in seven entities for the finest grouping level

```
{
  "odata.context":"$metadata#Sales(Customer(Country,Name),Product(Category(Nam
e),Name),Amount,Currency(Code))",
```



```
  "value": [
    { "odata.id": null, "Customer": { "Country": "USA", "Name": "Joe" },
      "Product": { "Category": { "Name": "Non-Food" }, "Name": "Paper" },
      "Amount": 1, "Currency": { "Code": "USD" }
    },
    ...
```

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

```
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Product": { "Category": { "Name": "Food" }, "Name": "Sugar" },
      "Amount": 2, "Currency": { "Code": "USD" }
    },
    ...
```

six without product name

```
    { "odata.id": null, "Customer": { "Country": "USA", "Name": "Joe" },
      "Product": { "Category": { "Name": "Food" } },
      "Amount": 6, "Currency": { "Code": "USD" }
    },
    ...
```

and four with neither customer nor product name

```
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Product": { "Category": { "Name": "Food" } },
      "Amount": 14, "Currency": { "Code": "USD" }
    },
    ...
  ]
}
```

3.11 Transformation `filter`

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 entities for which this expression evaluates to `true`.

Example 25:

```
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.

3.12 Transformation `expand`

The `expand` transformation takes a navigation property path as its first parameter and a Boolean expression as its second parameter that could also be passed as a `$filter` system query option to the set of related entities.

The result set is the input set with the specified navigation property expanded by those related entities for which the Boolean expression evaluates to `true`.

Example 26:

```
GET ~/Customers?$apply=expand(Sales,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": []}
]
```

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

```
GET ~/Customers?$expand=Sales
```

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 if the path contains more than one navigation property.

3.13 Transformation `search`

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 27: 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 `filter` has the same structure as the input set.

3.14 Filter Function `isdefined`

Properties that are not explicitly mentioned in `aggregate` or `groupby` are considered to have been *aggregated away* and are treated as having the null value in `$filter` expressions.

The filter function `isdefined` can be used to determine whether a property has been aggregated away. It takes a single-valued property path as its only argument 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 28: assuming that free-text search on `Sales` takes the related product name into account,

```
GET ~/Sales?$apply=aggregate(Amount)&$filter=isdefined(Product)
```

results in

```
{
```



```
  "odata.context": "$metadata#Sales(Amount)",
  "value": []
}
```

3.15 Evaluating `$apply`

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

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.

Providers MAY support `$count`, `$top` and `$skip` together with `rollup`, in which case rollup entities and detail entities are treated identically.

If a provider cannot satisfy a request using `$apply`, it MUST respond with `501 Not Implemented` and a human-readable error message.

3.16 ABNF for Extended URL Conventions

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

4 Representation of Aggregated Entities

Aggregated entities are based on the structure of the individual entities 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 entities.

Dynamic properties introduced with the keyword `as` are represented as defined by the response format.

Aggregated entities MAY be transient or persistent. Transient entities don't possess an edit link or read link, and in the JSON representation are marked with `"odata.id": null`. 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]**, sections 4.1 and 4.2 MUST be met.

Example 29: looking again to the sample request for getting sales amounts per product and country presented in section 3.10.1 (Example 21):

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

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

```
{
  "odata.context":"$metadata#Sales(Customer(Country),Product(Name),Amount)",
  "value": [
    { "odata.id": null,
      "Customer": { "Country": "Netherlands" },
      "Product": { "Name": "Paper" },
      "Amount": 3
    },
    ...
  ]
}
```


5 Cross-Joins and Aggregation

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

In some cases, however, requests may span entity sets with no predefined associations. Such requests can be sent to the entity container instead of an individual entity set. The entity container acts as an entity set that has implicit navigation properties with cardinality to-one to each entity set it contains, and queries across entity sets can be formulated using these implicit navigation properties.

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 pose queries across entity sets based on other join conditions.

Example 30: 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 31: using the `$crossjoin` resource for aggregate queries

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

results in

```
{
  "odata.context": "$metadata#Collection(Edm.ComplexType)",
  "value": [
    { "Products": { "Name": "Coffee" }, "Sales": { "Amount": 12 } },
    { "Products": { "Name": "Paper"  }, "Sales": { "Amount":  8 } },
    { "Products": { "Name": "Sugar"  }, "Sales": { "Amount":  4 } }
  ]
}
```

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

6 Vocabulary for Data Aggregation

The following terms are defined in the vocabulary for data aggregation **[OData-VocAggr].** They are included in an entity model schema with the alias `Aggregation`.

The term `SupportsApply` describes the aggregation capabilities and lists the properties that can be used for aggregate queries on the annotated entity set or container in the `$apply` system query option. It has a structured type with the following properties:

- `SupportsCount` indicates whether the virtual property `$count` can be used in `aggregate`.
- The `SupportedTransformations` collection lists all supported set transformations. Allowed values are the names of the standard transformations defined in chapter 3: `aggregate`, `topcount`, `topsum`, `toppercent`, `bottomcount`, `bottomsum`, `bottompercent`, `identity`, `concat`, `groupby`, `filter`, and `expand`, or a namespace-qualified name identifying a service-defined bindable function. If `SupportedTransformations` is omitted the server supports all transformations.
- The `AggregatableProperties` collection lists all properties that can be used in the `aggregate` transformation. If it is omitted the server supports all properties not listed as `GroupableProperties` (including the `DependentProperties`) in the `aggregation` transformation.
- The `DynamicAggregatableProperties` collection lists all dynamic properties that can be used in the `aggregate` transformation.
- The `GroupableProperties` collection lists all properties that can be used in the `groupby` transformation. If it is omitted then all properties not listed as `AggregatableProperties` can be used in the `groupby` transformation.
- Properties can appear in both collections. At least one of these collections SHOULD be provided; if both are omitted the `AggregatableProperties` and `GroupableProperties` are unspecified.
- `SupportsRollup` specifies to which extent the server supports rollup hierarchies in a `groupby` transformation.

6.1 Aggregatable Properties

A property that can be used in the `aggregate` transformation is described with the complex type `AggregatableProperty` which has the following properties:

- The `Name` contains the path to the aggregatable property.
- The `DefaultAggregationMethod` is either one of the standard aggregation methods `sum`, `min`, `max`, `average`, and `countdistinct`, or a namespace-qualified name identifying a provider-specific function, see section 3.1.2. It MUST appear in the list of supported aggregation methods.
- The `SupportedAggregationMethods` collection lists all supported aggregation methods. If the collection of supported aggregation methods is null or not specified the aggregatable property MAY only be used without specifying an aggregation method.

This collection is of a complex type with the properties:

- The `Name` follows the same rule as the `DefaultAggregationMethod`.
- The `RequiredGroupingProperties` collection lists properties that MUST be specified in the first parameter of `groupby` when using the aggregatable property in `aggregate`. Typical candidates for required grouping properties are properties that contain the currency code for a monetary amount or the unit of measure for a measured quantity (these SHOULD also be annotated using the Measures vocabulary, see **[OData-VocMeas])**.

6.2 Dynamic Aggregatable Properties

Some aggregated values are only defined for certain levels of aggregation. If represented as declared properties, they would not possess values for queries without `aggregate`, which would be somewhat confusing. Therefore, they can be represented as dynamic properties and are declared in an additional collection property of the `SupportsApply` annotation on entity set or entity container level. A dynamic aggregatable property is described by a complex type with the following properties:

- The `Name` must not collide with the names of declared properties or other dynamic aggregatable properties.
- The `Type` must be the qualified name of a primitive type or type definition in scope.
- The `SupportedGroupingProperties` collection lists the properties that define the maximum level of granularity for the dynamic aggregatable property. It only has a defined value of the set of grouping properties is a subset of or identical to the set of supported grouping properties.

Example 32: Sales forecasts are only available per month and country, and budget is only available per org unit and year. Sales forecasts are modeled as a dynamic aggregation property of the Sales entity set because it belongs there. For the budget, there's no appropriate entity set because there's no matching association between org units and time, so it is placed it at the entity container level.

```
<Annotations Target="SalesModel.Container">
  <Annotation Term="DataAggregation.SupportsApply">
    <Record>
      <PropertyValue Property="DynamicAggregatableProperties">
        <Collection>
          <Record>
            <PropertyValue Property="Name" String="Budget" />
            <PropertyValue Property="Type" String="Edm.Decimal" />
            <PropertyValue Property="SupportedGroupingProperties">
              <Collection>
                <PropertyPath>Time/Year</PropertyPath>
                <PropertyPath>SalesOranizations/Unit</PropertyPath>
              </Collection>
            </PropertyValue>
          </Record>
        </Collection>
      </PropertyValue>
      ...
    </Record>
  </Annotation>
</Annotations>

<Annotations Target="SalesModel.Container/Sales">
  <Annotation Term="DataAggregation.SupportsApply">
    <Record>
      <PropertyValue Property="DynamicAggregatableProperties">
        <Collection>
          <Record>
            <PropertyValue Property="Name" String="ForeCast" />
            <PropertyValue Property="Type" String="Edm.Decimal" />
            <PropertyValue Property="SupportedGroupingProperties">
              <Collection>
                <PropertyPath>Customer/Country</PropertyPath>
                <PropertyPath>Time/Month</PropertyPath>
                <PropertyPath>Time/Quarter</PropertyPath>
                <PropertyPath>Time/Year</PropertyPath>
              </Collection>
            </PropertyValue>
          </Record>
        </Collection>
      </PropertyValue>
    </Record>
  </Annotation>
```



```
</Annotations>
```

These dynamic aggregatable properties can now be used in the `aggregate` *transformation:*

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

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

6.3 Groupable Properties

A property that can be used in `groupby` is described by a complex type with the following properties:

- The `Name` contains the path to the groupable property.
- The `DependentProperties` collection lists properties that are functionally dependent on the groupable property and should not be used in `groupby` without this groupable property. In the example data, the customer `Name` could be listed as dependent on the customer `ID`. Providers MUST respond with `400 Bad Request` if a dependent property is used in the first parameter of `groupby` without its groupable property.

6.4 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.*

6.4.1 Leveled Hierarchy

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.

6.4.2 Recursive Hierarchy

A *recursive hierarchy* organizes the values of a single groupable property as nodes of a hierarchical 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 `ParentNodeProperty` contains the path to the identifier of the parent node.
- The optional `HierarchyLevelProperty` contains the path to a property that contains the level of the node in the hierarchy.
- 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 Filter Functions.

6.4.2.1 Hierarchy Filter Functions

For testing the position of a given entity instance in a recursive hierarchy annotated to the entity’s type, the vocabulary includes model functions that can be applied to any entity in `$filter` expressions:

- `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,
- `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,
- `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.

6.4.3 Examples

Example 33: 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="DataAggregation"
                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="DataAggregation.LeveledHierarchy"
                Qualifier="ProductHierarchy">
     <Record>
      <PropertyValue Property="Levels">
        <Collection>
          <String>Category/Name</String>
          <String>Name</String>
        </Collection>
      </PropertyValue>
     </Record>
    </Annotation>
   </Annotations>

   <Annotations Target="SalesModel.Time">
    <Annotation Term="DataAggregation.LeveledHierarchy"
                Qualifier="TimeHierarchy">
     <Record>
      <PropertyValue Property="Levels">
       <Collection>
         <String>Year</String>
         <String>Quarter</String>
         <String>Month</String>
       </Collection>
      </PropertyValue>
     </Record>
    </Annotation>
   </Annotations>

   <Annotations Target="SalesModel.SalesOrganization">
    <Annotation Term="DataAggregation.RecursiveHierarchy"
                Qualifier="SalesOrgHierarchy">
     <Record>
      <PropertyValue Property="NodeProperty" String="ID" />
      <PropertyValue Property="ParentNodeProperty"
```



```
                    String="Superordinate/ID" />
    </Record>
   </Annotation>
  </Annotations>
 </Schema>
</edmx:DataServices>
</edmx:Edmx>
```

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

Example 34: requesting all organizations below EMEA

```
GET ~/SalesOrganizations?
          $filter=$it/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 35: requesting just those organizations directly below EMEA

```
GET SalesOrganizations?$filter=$it/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 36: just the lowest-level organizations

```
GET SalesOrganizations?$filter=$it/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 37: the lowest-level organizations including their superordinate's ID


```
GET SalesOrganizations?$filter=$it/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 38: retrieving the sales IDs involving sales organizations from EMEA can be requested by

```
GET Sales?$select=ID
        &$filter=SalesOrganization/isdescendant(Hierarchy='SalesOrgHierarchy',
                                                Node='EMEA')
```

results in

```
{
  "odata.context": "$metadata#Sales(ID)",
  "value": [
    { "ID": 6 },
    { "ID": 7 },
    { "ID": 8 }
  ]
}
```

6.5 Actions and Functions on Aggregated Entities

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 39: 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 40: 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":
              "http://host/service/$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.

7 Examples

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

7.1 Distinct Values

Example 41:

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

results in

```
{
  "odata.context": "$metadata#Customers(Name)",
  "value": [
    { "odata.id": null, "Name": "Luc" },
    { "odata.id": null, "Name": "Joe" },
    { "odata.id": null, "Name": "Sue" }
  ]
}
```

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

Aggregation is also possible across related entities.

Example 42: customers that bought something

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

results in

```
{
  "odata.context": "$metadata#Sales(Customer(Name))",
  "value": [
    { "odata.id": null, "Customer": { "Name": "Joe" } },
    { "odata.id": null, "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 43:

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

results in

```
{
  "odata.context": "$metadata#Sales(Customer(Name,ID))",
  "value": [
    { "odata.id": null, "Customer": { "Name": "Joe", "ID": "C1" } },
    { "odata.id": null, "Customer": { "Name": "Sue", "ID": "C2" } },
    { "odata.id": null, "Customer": { "Name": "Sue", "ID": "C3" } }
```



```
  ]
}
```

This could also have been formulated as

```
GET ~/Sales?$apply=groupby((Customer))
           &$expand=Customer($select=Name,ID)
```

Grouping by a navigation property adds the 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`.

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 44: 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),
                          aggregate())
```

and results in

```
{
  "odata.context": "$metadata#Sales(Customer(Name,ID),Product(Name))",
  "value": [
    { "odata.id": null, "Customer": { "Name": "Joe", "ID": "C1" },
      "Product": { "Name": "Coffee"} },
    { "odata.id": null, "Customer": { "Name": "Joe", "ID": "C1" },
      "Product": { "Name": "Paper" } },
    { "odata.id": null, "Customer": { "Name": "Joe", "ID": "C1" },
      "Product: { "Name: "Sugar" } },
    { "odata.id": null, "Customer": { "Name": "Sue", "ID": "C2" },
      "Product: { "Name": "Coffee"} },
    { "odata.id": null, "Customer": { "Name": "Sue", "ID": "C2" },
      "Product": { "Name": "Paper" } },
    { "odata.id": null, "Customer": { "Name": "Sue", "ID": "C3" },
      "Product": { "Name": "Paper" } },
    { "odata.id": null, "Customer": { "Name": "Sue", "ID": "C3" },
      "Product": { "Name": "Sugar" } }
  ]
}
```

7.2 Provider-Controlled Aggregation

The aggregated values will be returned in the entity property whose values are aggregated, of course without changing its type.

Example 45:

```
GET ~/Products?$apply=groupby((Name),aggregate(Sales/Amount))
```

results in

```
{
  "odata.context": "$metadata#Products(Name,Sales(Amount))",
```



```
  "value": [
    { "odata.id": null, "Name": "Coffee", "Sales": [ { "Amount": 12 } ] },
    { "odata.id": null, "Name": "Paper",  "Sales": [ { "Amount":  8 } ] },
    { "odata.id": null, "Name": "Pencil", "Sales": [] },
    { "odata.id": null, "Name": "Sugar",  "Sales": [ { "Amount":  4 } ] }
  ]
}
```

Note that aggregation does not alter the cardinality of the Sales navigation property, and that it always returns an array with at most one object. If there are no "base" entities to be aggregated, the array is empty.

Example 46: 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=groupby((Name,Sales/Currency/Code),
                              aggregate(Sales/Amount))
```

results in

```
{
  "odata.context": "$metadata#Products(Name,Sales(Amount,Currency(Code)))",
  "value": [
    { "odata.id": null, "Name": "Coffee",
      "Sales": [ { "Amount": 12, "Currency": { "Code": "USD" } } ] },
    { "odata.id": null, "Name: "Paper",
      "Sales": [ { "Amount":  3, "Currency": { "Code": "EUR" } },
                 { "Amount":  5, "Currency": { "Code": "USD" } } ] },
    { "odata.id": null, "Name: "Pencil",
      "Sales": [] },
    { "odata.id": null, "Name: "Sugar",
      "Sales": [ { "Amount":  2, "Currency": { "Code": "EUR" } },
                 { "Amount":  2, "Currency": { "Code": "USD" } } ] }
  ]
}
```

Note that associations are "expanded" in a left-outer-join fashion, starting from the target of the aggregation request, before grouping the entities for aggregation. Afterwards the results are "folded back" to match the cardinality.

Example 47:

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

returns the different products sold per country:

```
{
  "odata.context": "$metadata#Customers(Country,Sales(Product(Name)))",
  "value": [
    { "odata.id": null, "Country": "Netherlands",
       "Sales": [ { "Product": { "Name": "Paper"  },
                    { "Product": { "Name": "Sugar"  } ] },
    { "odata.id": null, "Country": "USA",
      "Sales": [ { "Product": { "Name": "Coffee" },
                 { "Product": { "Name": "Paper"  },
                 { "Product": { "Name": "Sugar"  } ] }
  ]
}
```

7.3 Consumer-Controlled Aggregation

Instead of using the provider-defined default aggregation behavior, the consumer may specify one of the predefined aggregation methods `min`, `max`, `sum`, `average`, and `countdistinct`.

Example 48:

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

results in

```
{
  "odata.context": "$metadata#Sales(Customer(Country),TotalAmount)",
  "value": [
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "TotalAmount":  5 },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "TotalAmount": 19 }
  ]
}
```

If no alias is specified, the aggregated values will be returned in the entity property whose values are aggregated, of course without changing its type.

Example 49:

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

results in

```
{
  "odata.context": "$metadata#Sales(Customer(Country),AverageAmount)",
  "value": [
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Amount": 1.6666667 },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Amount": 3.8 }
  ]
}
```

In the example above Amount is an `Edm.Decimal` with `Scale="variable"`. If it were an integer or had a fixed `Scale`, the aggregated values would have been rounded accordingly, see section 3.1.2.4.

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 50:

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

results in

```
{
  "odata.context": "$metadata#Products(Name,SalesCount)",
  "value": [
    { "odata.id": null, "Name": "Coffee", "SalesCount": 2 },
    { "odata.id": null, "Name": "Paper",  "SalesCount": 4 },
    { "odata.id": null, "Name": "Pencil", "SalesCount": 0 },
    { "odata.id": null, "Name": "Sugar",  "SalesCount": 2 }
  ]
}
```

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

Example 51: the result properties for Sales/$count and Sales/Amount are placed differently


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

results in

```
{
  "odata.context": "$metadata#Products(Name,SalesCount,Sales(TotalAmount))",
  "value": [
    { "odata.id": null, "Name": "Coffee", "SalesCount": 2,
      "Sales": [ { "TotalAmount": 12 } ] },
    { "odata.id": null, "Name": "Paper",  "SalesCount": 4,
      "Sales": [ { "TotalAmount":  8 } ] },
    { "odata.id": null, "Name": "Pencil", "SalesCount": 0,
      "Sales": [ { "TotalAmount":  0 } ] },
    { "odata.id": null, "Name": "Sugar",  "SalesCount": 2,
      "Sales": [ { "TotalAmount":  4 } ] }
  ]
}
```

7.4 Aliasing

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

Example 52:

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

results in

```
{
  "odata.context": "$metadata#Sales(Customer(Country),Amount,AvgAmt))",
  "value": [
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Amount":  5, "AvgAmt": 1.6666667 },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Amount": 19, "AvgAmt": 3.8 }
  ]
}
```

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

Example 53:

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

results in

```
{
  "odata.context": "$metadata#Products(Name,Sales(Amount,AvgAmt))",
  "value": [
    { "odata.id":null, "Name":"Coffee", "Sales":[{"Amount":12,"AvgAmt":6}] },
    { "odata.id":null, "Name":"Paper",  "Sales":[{"Amount": 8,"AvgAmt":2}] },
    { "odata.id":null, "Name":"Pencil", "Sales":[] },
    { "odata.id":null, "Name":"Sugar",  "Sales":[{"Amount": 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 54:


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

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,TotalSales)",
  "value": [
    { "odata.id": null, "Amount": 1, "TotalSales": 2 },
    { "odata.id": null, "Amount": 2, "TotalSales": 6 },
    { "odata.id": null, "Amount": 4, "TotalSales": 8 },
    { "odata.id": null, "Amount": 8, "TotalSales": 8 }
  ]
}
```

7.5 Combining Transformations per Group

Example 55: 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))
                      /groupby((Customer/Country,Currency/Code),
                               topcount(1,Amount)),
                    groupby((Customer/Country,Currency/Code),
                            aggregate(Amount)))
```

results in

```
{
  "odata.context":
     "$metadata#Sales(Customer(Country),Product(Name),Amount,Currency(Code))",
  "value": [
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Product":  { "Name": "Coffee" },
      "Amount": 12, "Currency": { "Code": "USD" }
    },
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Product":  { "Name": "Paper" },
      "Amount":  3, "Currency": { "Code": "EUR" }
    },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Amount": 19, "Currency": { "Code": "USD" }
    },
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Amount":  5, "Currency": { "Code": "EUR" }
    }
  ]
}
```

Example 56: 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))
```

results in

```
{
  "odata.context":
     "$metadata#Sales(Customer(Country),Product(Name),Amount,Currency(Code))",
  "value": [
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Product": { "Name": "Paper" },
```



```
      "Amount":  3, "Currency": { "Code": "EUR" }
    },
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Product": { "Name": "Sugar" },
      "Amount":  2, "Currency": { "Code": "EUR" }
    },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Product": { "Name": "Coffee" },
      "Amount": 12, "Currency": { "Code": "USD" }
    },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Product": { "Name": "Paper" },
      "Amount":  5, "Currency": { "Code": "USD" }
    }
  ]
}
```

7.6 Applying Model Functions as Set Transformations

Example 57: 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 argument and 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.

The request

```
GET ~/Sales?$apply=groupby((Customer/Country,Product/Name),aggregate(Amount))
                  /groupby((Customer/Country),
                           Self.TopCountAndBalance(Count=1,Property='Amount'))
```

returns the entities shown in the request result in the previous section plus the following entities

```
{
  "odata.context": "$metadata#Sales(Customer(Country),Product(Name),Amount)",
  "value": [
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Product": { "Name": "Paper" }, "Amount":  3 },
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "Product": { "Name": "**Other**" }, "Amount":  2 },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Product": { "Name": "Coffee" }, "Amount": 12 },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "Product": { "Name": "**Other**" }, "Amount":  5 }
  ]
}
```

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.

7.7 Controlling Aggregation per Rollup Level

Consumers may specify a different aggregation method per level for every property passed to `rollup` as a hierarchy level below the root level.

Example 58: 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 SumAmount))
                  /groupby((Product/ID,Product/Name),
                           aggregate(SumAmount with average as AverageAmount))
```

Using `from`:

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

Example 59: for an aggregate entity set listing the total sales amounts per customer and country, the rollup shall produce additional entities 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 currency☺)

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

results in

```
{
  "odata.context":
"$metadata#Sales(Customer(Country,ID),CustomerCountryAverage,Currency(Code))",
  "value": [
    { "odata.id": null, "Customer": { "Country": "USA", "ID": "C1" },
      "CustomerCountryAverage":   7, "Currency": { "Code": "USD" }
    },
    { "odata.id": null, "Customer": { "Country": "USA", "ID": "C2" },
      "CustomerCountryAverage":  12, "Currency": { "Code": "USD" }
    },
    { "odata.id": null, "Customer": { "Country": "USA" },
      "CustomerCountryAverage": 9.5, "Currency": { "Code": "USD" }
    },
    { "odata.id": null, "Customer": { "Country": "Netherlands", "ID": "C3" },
      "CustomerCountryAverage":   5, "Currency": { "Code": "EUR" }
    },
    { "odata.id": null, "Customer": { "Country": "Netherlands" },
      "CustomerCountryAverage":   5, "Currency": { "Code": "EUR" }
    },
    { "odata.id": null,
      "CustomerCountryAverage": 9.5, "Currency": { "Code": "USD" }
    },
    { "odata.id": null,
      "CustomerCountryAverage":   5, "Currency": { "Code": "EUR" }
    }
  ]
}
```

7.8 Transformation Sequences

Applying aggregation first covers the most prominent use cases, yet it is insufficient to answer more sophisticated questions like "how much money is earned with small sales", which 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 60:

```
GET ~/Sales?$apply=filter(Amount le 1)/aggregate(Amount)
```

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

```
{
  "odata.context": "$metadata#Sales(Amount)",
  "value": [
    { "odata.id": null, "Amount": 2 }
  ]
}
```

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

Example 61:

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

results in

```
{
  "odata.context": "$metadata#Sales(Product(Name),Amount)",
  "value": [
    { "odata.id": null, "Amount": 4, "Product": { "Name": "Paper" } },
    { "odata.id": null, "Amount": 4, "Product": { "Name": "Sugar" } }
  ]
}
```

Example 62: Revisiting the Example 11 in section 3.1.3 for using the `from` keyword with the `aggregate` function, the request

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

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

```
GET ~/Sales?$apply=groupby((Time),aggregate(Amount)
                 /aggregate(Amount 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 63: getting the population per country with

```
GET ~/Cities?$apply=groupby((Continent/Name,Country/Name),
                            aggregate(Population))
```

results in

```
{
  "odata.context":
                "$metadata#Cities(Continent(Name),Country(Name),Population)",
  "value": [
    { "odata.id": null, "Continent": { "Name": "Asia" },
      "Country": { "Name": "China" }, "Population": 692.580.000 },
    { "odata.id": null, "Continent": { "Name": "Asia" },
      "Country": { "Name": "India" }, "Population": 390.600.000 },
    ...
  ]
}
```


Example 64: all countries with megacities and their continents

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

Example 65: 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))
                  /filter(Population ge 10000000)
                  /concat(identity,
                          groupby((Continent/Name),
                                  aggregate(Population)))
```

-- OR –

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

Example 66: 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))
                  /concat(filter(Population ge 10000000),
                          groupby((Continent/Name),
                                  aggregate(Population)))
```

Example 67: filter the base set and filter related items before aggregation

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


8 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 `SupportsApply` annotation.

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.

Appendix A. Acknowledgments

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

Appendix B. Revision History

Revision	Date	Editor	Changes Made
Working Draft 01	2012-11-12	Ralf Handl	Translated contribution into OASIS format
Committee Specification Draft 01	2013-07-25	Ralf Handl Hubert Heijkers Gerald Krause Michael Pizzo Martin Zurmuehl	Switched to pipe-and-filter-style query language based on composable set transformations Fleshed out examples and addressed numerous editorial and technical issues processed through the TC Added Conformance section

