OData Extension for Data Aggregation
Version 4.0

Committee Specification Draft 0402 /
Public Review Draft 0402

25 July 2013

09 January 2014

Specification URIs
This version:
http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd02/odata-data-aggregation-ext-v4.0-csprd02.docx (Authoritative)
http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd02/odata-data-aggregation-ext-v4.0-csprd02.html
http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd02/odata-data-aggregation-ext-v4.0-csprd02.pdf

Previous 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

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

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/csprd042/abnf/
- OData Aggregation Vocabulary: http://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csprd042/vocabularies/

Related work:
This specification is related to:
- **OData Version 4.0.** Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. A multi-part Work Product that includes:

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]
Notices

Copyright © OASIS Open 2014. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the “OASIS IPR Policy”). The full Policy 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.........................................................................................................................6  
1.1 Terminology.......................................................................................................................6  
1.2 Normative References .......................................................................................................6  
1.3 Non-Normative References ...............................................................................................6  
1.4 Typographical Conventions .............................................................................................6  
2 Overview...............................................................................................................................7  
2.1 Definitions ..........................................................................................................................7  
2.2 Example Data Model .........................................................................................................7  
2.3 Example Data ....................................................................................................................10  
2.4 Example Use Cases ...........................................................................................................12  
3 System Query Option $apply................................................................................................14  
3.1 Transformation aggregate...............................................................................................14  
3.1.1 Keyword as.....................................................................................................................15  
3.1.2 Keyword with..................................................................................................................16  
3.1.3 Aggregation Methods .................................................................................................16  
3.1.4 Keyword from...............................................................................................................19  
3.1.5 Virtual Property $count...............................................................................................20  
3.2 Transformation topcount...............................................................................................20  
3.3 Transformation topsum.................................................................................................21  
3.4 Transformation toppercent..............................................................................................21  
3.5 Transformation bottomcount...........................................................................................22  
3.6 Transformation bottomsum.............................................................................................23  
3.7 Transformation bottompercent.......................................................................................23  
3.8 Transformation identity.................................................................................................24  
3.9 Transformation concat....................................................................................................24  
3.10 Transformation groupby...............................................................................................24  
3.10.1 Simple Grouping..........................................................................................................25  
3.10.2 Grouping with rollup and $all....................................................................................27  
3.11 Transformation filter......................................................................................................28  
3.12 Transformation expand...................................................................................................29  
3.13 Transformation search...................................................................................................30  
3.14 Filter Function isdefined...............................................................................................30  
3.15 Evaluating $apply...........................................................................................................30  
3.16 ABNF for Extended URL Conventions ........................................................................31  
4 Representation of Aggregated Instances ........................................................................32  
5 Cross-Joins and Aggregation .............................................................................................33  
6 Vocabulary for Data Aggregation .......................................................................................34  
6.1 Aggregation Capabilities ...............................................................................................34  
6.2 Property Annotations ......................................................................................................35  
6.2.1 Groupable Properties .................................................................................................35  
6.2.2 Aggregatable Properties .............................................................................................35  
6.2.3 Custom Aggregates.....................................................................................................35
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

See the link in "Related work" section on cover page.

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.

See link in "Related work" section on cover page.

See link in "Related work" section on cover page.

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

See link in "Related work" section on cover page.


1.3 Non-Normative References


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 express explicitly 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 using an aggregation method.
- **Aggregation Method** – a method that can be used to aggregate an aggregatable property or expression
- **Standard Aggregation Method** – one of the standard aggregation methods: sum, min, max, average, and count distinct
- **Custom Aggregation Method** – a custom aggregation method that can be applied to expressions of a specified type
- **Custom Aggregate Groupable** – a dynamic property that can appear in an aggregate clause
- **Groupable Property** – a property whose values can be used to group entities or complex type instances 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

*Example 2:* The following diagram shows the terms defined in the section above applied to a simple model that is used throughout this document.
The Amount property in the Sales entity type is an aggregatable property, and the properties of the related entity types are groupable. These can be arranged in four hierarchies:

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

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.

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

*Example 3:* The following sample data will be used to further illustrate the capabilities introduced by this extension.
### Time
<table>
<thead>
<tr>
<th>Date</th>
<th>Month</th>
<th>Quarter</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-01-01</td>
<td>2012-01</td>
<td>2012-1</td>
<td>2012</td>
</tr>
<tr>
<td>2012-04-01</td>
<td>2012-04</td>
<td>2012-2</td>
<td>2012</td>
</tr>
<tr>
<td>2012-04-10</td>
<td>2012-04</td>
<td>2012-2</td>
<td>2012</td>
</tr>
</tbody>
</table>

### Customers
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Joe</td>
<td>USA</td>
</tr>
<tr>
<td>C2</td>
<td>Sue</td>
<td>USA</td>
</tr>
<tr>
<td>C3</td>
<td>Sue</td>
<td>Netherlands</td>
</tr>
<tr>
<td>C4</td>
<td>Luc</td>
<td>France</td>
</tr>
</tbody>
</table>

### Products
<table>
<thead>
<tr>
<th>ID</th>
<th>Category</th>
<th>Name</th>
<th>Color</th>
<th>TaxRate</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>PG1</td>
<td>Sugar</td>
<td>White</td>
<td>0.06</td>
</tr>
<tr>
<td>P2</td>
<td>PG1</td>
<td>Coffee</td>
<td>Brown</td>
<td>0.06</td>
</tr>
<tr>
<td>P3</td>
<td>PG2</td>
<td>Paper</td>
<td>White</td>
<td>0.14</td>
</tr>
<tr>
<td>P4</td>
<td>PG2</td>
<td>Pencil</td>
<td>Black</td>
<td>0.14</td>
</tr>
</tbody>
</table>

### Categories
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG1</td>
<td>Food</td>
</tr>
<tr>
<td>PG2</td>
<td>Non-Food</td>
</tr>
</tbody>
</table>

### SalesOrganizations
<table>
<thead>
<tr>
<th>ID</th>
<th>Superordinate</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales</td>
<td>Corporate Sales</td>
<td></td>
</tr>
<tr>
<td>US Sales</td>
<td>US</td>
<td></td>
</tr>
<tr>
<td>US West US West</td>
<td>US West</td>
<td></td>
</tr>
<tr>
<td>US East US East</td>
<td>US East</td>
<td></td>
</tr>
<tr>
<td>EMEA Sales EMEA EMEA Central</td>
<td>EMEA EMEA Central</td>
<td></td>
</tr>
</tbody>
</table>

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

### Currencies
<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>US Dollar</td>
</tr>
<tr>
<td>EUR</td>
<td>Euro</td>
</tr>
</tbody>
</table>
2.4 Example Use Cases

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

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

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

<table>
<thead>
<tr>
<th>Customer/Country</th>
<th>Customer/Name</th>
<th>Product/Category</th>
<th>Product/Name</th>
<th>Amount</th>
<th>Currency /Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>Joe</td>
<td>Food</td>
<td>Sugar</td>
<td>1</td>
<td>USD</td>
</tr>
<tr>
<td>USA</td>
<td>Joe</td>
<td>Food</td>
<td>Coffee</td>
<td>2</td>
<td>USD</td>
</tr>
<tr>
<td>USA</td>
<td>Sue</td>
<td>Food</td>
<td>Coffee</td>
<td>4</td>
<td>USD</td>
</tr>
<tr>
<td>Netherlands</td>
<td>Sue</td>
<td>Non-Food</td>
<td>Paper</td>
<td>3</td>
<td>EUR</td>
</tr>
<tr>
<td>USA</td>
<td>Sue</td>
<td>Non-Food</td>
<td>Paper</td>
<td>2</td>
<td>EUR</td>
</tr>
<tr>
<td>USA</td>
<td>Sue</td>
<td>Non-Food</td>
<td>Paper</td>
<td>3</td>
<td>EUR</td>
</tr>
</tbody>
</table>

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:
<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>Category</th>
<th>Item</th>
<th>Quantity</th>
<th>Currency</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>Sue</td>
<td>Non-Food</td>
<td>Paper</td>
<td>4</td>
<td>USD</td>
</tr>
<tr>
<td>Netherlands</td>
<td>Sue</td>
<td>Food</td>
<td>Sugar</td>
<td>2</td>
<td>EUR</td>
</tr>
<tr>
<td>Netherlands</td>
<td>Sue</td>
<td>Non-Food</td>
<td>Paper</td>
<td>3</td>
<td>EUR</td>
</tr>
<tr>
<td>USA</td>
<td>Food</td>
<td>Sugar</td>
<td>2</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Food</td>
<td>Coffee</td>
<td>12</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Non-Food</td>
<td>Paper</td>
<td>5</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>Netherlands</td>
<td>Food</td>
<td>Sugar</td>
<td>2</td>
<td>EUR</td>
<td></td>
</tr>
<tr>
<td>Netherlands</td>
<td>Non-Food</td>
<td>Paper</td>
<td>1</td>
<td>EUR</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Joe</td>
<td>Food</td>
<td>6</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Joe</td>
<td>Non-Food</td>
<td>1</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Sue</td>
<td>Food</td>
<td>8</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Sue</td>
<td>Non-Food</td>
<td>4</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>Netherlands</td>
<td>Sue</td>
<td>Food</td>
<td>2</td>
<td>EUR</td>
<td></td>
</tr>
<tr>
<td>Netherlands</td>
<td>Sue</td>
<td>Non-Food</td>
<td>3</td>
<td>EUR</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Food</td>
<td></td>
<td>14</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>Non-Food</td>
<td></td>
<td>5</td>
<td>USD</td>
<td></td>
</tr>
<tr>
<td>Netherlands</td>
<td>Food</td>
<td></td>
<td>2</td>
<td>EUR</td>
<td></td>
</tr>
<tr>
<td>Netherlands</td>
<td>Non-Food</td>
<td></td>
<td>3</td>
<td>EUR</td>
<td></td>
</tr>
</tbody>
</table>

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 entityType, so the structure of the result fits into the data model of the service.
- does not necessarily preserve the number of instances in the result entries, as this will typically differ from the number of instances in the input entity set.
- does not necessarily guarantee that all properties of the result entityInstances 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], with dynamic properties representing the aggregate values. The parameters of set transformations allow specifying how the result entityInstances are constructed from the input entityInstances.

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 one or more aggregate expressions as parameters and returns a result set with a single entityInstance representing the aggregated value for all entityInstances in the input set.

In its simplest form, it takes one or more paths to properties as parameters. The resulting entity contains those properties (similar to An aggregate expression may be:

- an expression valid in a $select$filter system query option on the input set that results in a simple value, e.g. the path to an aggregatable property, with a specified aggregation method.
• a custom aggregate,
• any of the above, followed by a from expression,
• any of the above, enclosed in parentheses and prefixed with a navigation path to related entities,
• the virtual property $count.

Any aggregate expression that specifies an aggregation method MUST define an alias, and their values represent the for the resulting aggregated value of those properties. The resulting instance contains one dynamic property per parameter representing the aggregated value across all entities of instances within 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:

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 An expression resulting in a simple value and a custom aggregate corresponds to an item a dynamic property in a $select option. If the parameter property is they are 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, Aggregate expressions can define an alias MAY be specified with using 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 result set. The introduced dynamic property MUST always be ins added to the same entity type as containing the original property expression or custom aggregate. The alias MUST NOT collide with names of declared properties, custom aggregates, or other aliases in that entity type.

When an aggregation method is specified, an alias MUST be applied to the expression.

Example 5:

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

results in

```
{
  "odata.context": "$metadata#Sales(TotalAmount)",
  "value": [
    { "odata.id": null, "TotalAmount": 24 }
  ]
}
```
Example 6:

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:

```
GET ~/Sales?$apply=aggregate(Amount mul Product/TaxRate with sum 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 7:

```
GET ~/Products?$apply=aggregate(Sales(Amount mul Product/TaxRate with sum 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 itema dynamic property in a $select option that is nested in an $expand option for each navigation property in the path of the aliased propertyexpression.

3.1.2 Keyword with

The keyword with is used to apply an aggregation method to an aggregatable property. To override the default aggregation behavior, a property path MAY be, or expression. The property or expression being aggregated is followed by the keyword with and, followed by the name of the aggregation method to apply, followed by the keyword as and an alias.

3.1.3 Aggregation Methods

Values can be aggregated using the standard aggregation methods sum, min, max, average, and countdistinct, or with custom aggregation methods defined by the service. Aggregate expressions
containing an aggregation method MUST define an alias. The predefined aggregation methods are `sum`, `min`, `max`, `average`, and `countdistinct`.

Providers MAY define non-standard aggregation methods using annotations, see section 3.3 for the resulting aggregate value.

Custom 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.

### 3.1.2.13.1.3.1 Standard Aggregation Method `sum`

The standard aggregation method `sum` MAY be applied to a numeric property. It returns values to return the sum of the non-null values, and or null if all values there are no non-null.

If an alias is introduced with the `alias keyword`, the values. The provider MUST choose a single type for the property across all instances of that type in 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.

**Example 8:**

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 9:**

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

results in

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

### 3.1.2.23.1.3.2 Standard Aggregation Method `min`

The standard aggregation method `min` MAY be applied to an property values with a totally ordered domain. It returns the smallest of the non-null values, and or null if all values there are no non-null values.

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

**Example 9:**

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

results in

```
{ "@odata.context": "$metadata#Sales(MinAmount)",
  "value": [
    { "@odata.id": null, "MinAmount": 1 }
  ]
}
```
3.1.2.3.3 Standard Aggregation Method max

The standard aggregation method max MAY be applied to an property values with a totally ordered domain. It returns the largest of the non-null values, and null if all values are no non-null values.

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

Example 10:

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

results in

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

3.1.2.4.3.1.3.4 Standard Aggregation Method average

The standard aggregation method average MAY be applied to a numeric 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 no non-null values.

If an alias is introduced with the keyword the result property will have the same type as defined in the provider.

If no alias is introduced, that is capable of representing the aggregated property will contain the result values, either Edm.Double or Edm.Decimal with sufficient Precision and the result will be rounded according to the rounding rules defined in Scale.

Example 9:

Example 11:

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

results in

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

3.1.2.5.3.1.3.5 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 keyword and the result property MUST have type Edm.Decimal with Scale=“0” and sufficient Precision.

Example 12:

Example 13:
GET ~/Sales?$apply=aggregate(Product with countdistinct as DistinctProducts)

results in

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

The number of instances in the input set can be counted with the virtual property $count.

### 3.1.3.3.1.4 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 an expression across all properties not included in the result structure, other aggregation methods to be applied when aggregating away certain properties MAY be specified using the `from` keyword, followed by a property path, and optionally of a groupable property. Each groupable property MUST be followed by a `with` clause. If the `with` clause aggregate expression is omitted, a custom aggregate, in which case the provider-controlled default aggregation method defined behavior of the custom aggregate is applied:

```
aggregatableProperty [with aggregationMethod aggregateExpression as alias]
from groupableProperty1 [with aggregationMethod1]
...
from groupablePropertyn [with aggregationMethodn]
```

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 or determined by the aggregatable property) aggregate expression is used to aggregate away properties that don't have are not mentioned in a special aggregation method specified from clause 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 and using the method specified by the `from` clause.

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

```
groupby((groupableProperty1, groupableProperty2, ..., groupablePropertyn),
  aggregate((aggregatableProperty) aggregateExpression as tmpalias1))
groupby((groupableProperty1, groupableProperty2, ..., groupablePropertyn),
  aggregate((aggregatableProperty) aggregateExpression as tmpalias1 with
  aggregationMethod1 aggregateMethodn as tmpalias))
  as aggregatableProperty
...
groupby((groupableProperty1, groupableProperty2, ..., groupablePropertyn),
  aggregate((aggregatableProperty) aggregateMethodn as tmpaliasn)
  as aggregatableProperty)
```
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 13:

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

is equivalent to

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

and results in the average sales volume per day

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

3.1.43.1.5 Virtual Property $count

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

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

Example 14:

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

results in

```plaintext
{
  "@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 instances 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on entities instances of the input set and MUST result in a primitive numeric value.

The transformation retains the number of entities instances 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 service MUST impose a stable ordering of the input set is additionally taken into account returned instances.

Example 15:

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

results in

```json
{
  "@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 instances 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on entities instances of the input set and MUST result in a primitive numeric value.

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

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering of the input set is additionally taken into account returned instances.

Example 16:

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

results in

```json
{
  "@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 toppercen

The toppercen transformation takes two parameters.

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

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

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering of the input set before determining the result set.

Example 17:

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

results in

```json
{
  "@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 instances 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on entities instances of the input set and MUST result in a primitive numeric value.

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

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering of the input set before determining the result set.

Example 18:

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

results in

```json
{
  "@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 instances 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on entities instances of the input set and MUST result in a primitive numeric value.

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

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering of the input set is additionally taken into account returned instances.

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 instances 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on entities instances of the input set and MUST result in a primitive numeric value.

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

In case the value of the second expression is ambiguous, the service MUST impose a stable ordering of the input set is additionally taken into account returned instances.

Example 20:

```
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 21:**

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

**Example 20:**

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

results in

```
{
    "@odata.context": "$metadata#Sales",
    "value": [
        { "ID": 4, "Amount": 8, ... },
        { "ID": 3, "Amount": 4, ... },
        { "ID": 7, "@odata.context": "$metadata#Sales(Amount=1,...),
        { "ID": 7, "$entity", "Amount": 1,...24 }
    ]
}
```

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

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

### 3.10 Transformation groupby

The groupby transformation takes one or two parameters and
1. Splits the entities of the initial set into subsets, transforms each where all instances in a subset have the same values for the grouping properties specified in the first parameter, and then concatenates the transformed subsets. It takes two parameters: grouping properties specified in the first parameter specifying how the input parameter.

2. Applies set is split into subsets, the second how transformations to each subset is transformed. This transformation may remove properties from the input set or add according to the second parameter, resulting in a new properties to it (see ) as well as the number of entities it contains set of potentially different structure and cardinality.

3. Ensures that the instances in the result set contain all grouping properties with the correct values for the group.

4. Concatenates the intermediate result sets into one result set.

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

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

Example 23:

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

results in

```json
{ "@odata.context": "$metadata#Sales(Customer(Country),Product( Name),Amount_Total)" , "value": [ { "@odata.id": null, "Customer": { "Country": "Netherlands" }, "Product": { "Name": "Paper" }, "Amount_Total": 3 }, { "@odata.id": null, "Customer": { "Country": "Netherlands" }, "Product": { "Name": "Sugar" }, "Amount_Total": 2 }, { "@odata.id": null, "Customer": { "Country": "USA" }, "Product": { "Name": "Coffee" }, "Amount_Total": 12 }, { "@odata.id": null, "Customer": { "Country": "USA" }, "Product": { "Name": "Paper" }, "Amount_Total": 5 }, { "@odata.id": null, "Customer": { "Country": "USA" }, "Product": { "Name": "Sugar" }, "Total": 2 } ] }
```
The second parameter can be omitted to request distinct value combinations of the grouping properties.

**Example 24:**

```plaintext
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

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

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

```plaintext
GET ~/Sales?$expand=Product($select=Name).$select=Amount
```

Combined with `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:** Both requests are equivalent

```plaintext
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 }
  
  ]
```

A `groupby` transformation affects the structure of the result set. similar to `@select where` each grouping property corresponds to an item in a `@select option-clause`. Grouping properties that specify navigation properties are automatically expanded, and the specified properties of that is nested in an `@expand option` for each navigation property in the path of the grouping correspond to properties specified in a `@select expand option on the expanded navigation` property. The set transformations used specified in the second parameter of `groupby` furthermore affect the structure as described for `everyeach` transformation. For example, the `aggregate transformation` adds properties for each aggregate expression.
### 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 `entitiesinstances`.

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 single-valued property paths and define consecutively finer-grained levels of the hierarchy. This unnamed hierarchy is used for grouping `entitiesinstances`.

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

```
(rollup(p_{1,1}, p_{1,2}), rollup(p_{2,1}, p_{2,2}, p_{2,3}))
```

will result in the six groupings

<table>
<thead>
<tr>
<th>p_{1,1}</th>
<th>p_{1,2}</th>
<th>p_{2,1}</th>
<th>p_{2,2}</th>
<th>p_{2,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_{1,1}</td>
<td>p_{1,2}</td>
<td>p_{2,1}</td>
<td>p_{2,2}</td>
<td>p_{2,3}</td>
</tr>
<tr>
<td>p_{1,1}</td>
<td>p_{1,2}</td>
<td>p_{2,1}</td>
<td>p_{2,2}</td>
<td></td>
</tr>
<tr>
<td>p_{1,1}</td>
<td>p_{1,2}</td>
<td>p_{2,1}</td>
<td>p_{2,2}</td>
<td></td>
</tr>
<tr>
<td>p_{1,1}</td>
<td>p_{1,2}</td>
<td>p_{2,1}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Example 26: answering the second question in section 2.4

```
GET ~/Sales?$apply=groupby((rollup(Customer/Country, Customer/Name),
                           rollup(Product/Category/Name, Product/Name),
                           Currency/Code),
                           aggregate(Amount with sum as Total))
```
results in seven entities for the finest grouping level

```
{
    "@odata.context":"$metadata#Sales(Customer(Country,Name),Product(Category(Name)),AmountTotal,Currency(Code))",
    "value": [
        {
            "@odata.id": null,
            "Customer": { "Country": "USA", "Name": "Joe" },
            "Product": { "Category": { "Name": "Non-Food" }, "Name": "Paper" },
            "AmountTotal": 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" },
    "AmountTotal": 2,
    "Currency": { "Code": "USD" }
},
,...
```

six without product name

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

and four with neither customer nor product name

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

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

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 instances for which this expression evaluates to true.

Example 27:

```
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 expand system query option as its first parameter. The optional second parameter can be either a filter transformation that will be applied to the set of related entities or an expand transformation. An arbitrary number of expand transformations can be passed as additional parameters to achieve multi-level expansion.

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

Example 28:

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

results in

```
{
   "@odata.context": "$metadata#Customers",
   "value": [
      {
         "ID": "C1", "Name": "Joe", "Country": "USA",
         "Sales": [{ "ID": 3, "Amount": 4, ... }],
      },
      {
         "ID": "C2", "Name": "Sue", "Country": "USA",
         "Sales": [{ "ID": 4, "Amount": 8, ... },
         { "ID": 5, "Amount": 4, ... }],
      },
      { "ID": "C3", "Name": "Sue", "Country": "Netherlands", "Sales": []},
      { "ID": "C4", "Name": "Luc", "Country": "France", "Sales": []}
   ]
}
```

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 for the optional nested expand transformations.

Example 29: nested expand transformations

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

results in

```
{
   "@odata.context": "$metadata#Customers",
   "value": [
      { "ID": "PG1", "Name": "Food",
         "Products": [
            { "ID": "P1", "Name": "Sugar", "Color": "White", "Sales": [] },
            { "ID": "P2", "Name": "Coffee", "Color": "Brown",
            "Sales": [{ "ID": 3, "Amount": 4, ... },
             { "ID": 4, "Amount": 8, ... } ] }
         ],
      },
      { "ID": "PG2", "Name": "Non-Food",
         "Products": [
            { "ID": "P3", "Name": "Paper", "Color": "White",
             "Sales": [{ "ID": 5, "Amount": 4, ... },
             { "ID": 8, "Amount": 2, ... } ] },
            { "ID": "P4", "Name": "Pencil", "Color": "Black", "Sales": [] }
         ]
   ]
}
```
## 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 30: 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 search 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 parameter and returns true if the property has a defined value for the aggregated entity. A property with a defined value can still have the null value; it can represent a grouping of null values, or an aggregation that results in a null value.

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

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

results in

```
{
  "@odata.context": "$metadata#Sales(AmountTotal)",
  "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 on the output set 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, i.e. instances for which a property has been rolled up appear before instances that have a null value for that property when ordering ascending.
Providers MAY support $count, $top and $skip together with rollup, in which case rollup entities and detail entities are treated identically. instances are counted identically to detail instances, i.e. $skip=5 skips the first five instances, independently of whether some of them are rollup entities.

If a provider cannot satisfy a request using $apply, 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 through an alias or with the keyword custom aggregates 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 32: looking again to the sample request for getting sales amounts per product and country presented in section 3.10.1 (Example 23):

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

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

```
{
 "@odata.context":"$metadata#Sales(Customer(Country),Product(Name),AmountTotal)",
 "value": [
  { "@odata.id": null,
    "Customer": { "Country": "Netherlands" },
    "Product": { "Name": "Paper" },
    "AmountTotal": 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 need to span entity sets with no predefined associations. Such requests can be sent to the entity container special resource $crossjoin instead of an individual entity set. The cross join of a list of entity sets is the Cartesian product of the listed entity container acts sets, represented as an entity sets collection of complex type instances that have implicit have a navigation properties with cardinality to-one to for each participating entity set it contains, and queries across entity sets can be formulated using these implicit navigation properties. See [OData-URL] for details.

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

Example 33: 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 ~/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 34: using the $crossjoin resource for aggregate queries

```
GET ~/Products/Sales
    ?$apply=filter(Products/ID eq Sales/ProductID)
    |/groupby((Products/Name), aggregate(Sales/Amount)).
    aggregate(Sales(Amount with sum as Total)))
```

```
results in

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
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.

6.1 Aggregation Capabilities

The term `SupportsApply` can be applied to an entity container or to structured types and describes the aggregation capabilities and lists of the properties that can be used for aggregate queries on entity container or of collections of instances of the annotated entity set or container in the system query option. It has a structured types. If present, it implies that instances of the annotated structured type, or of structured types used in the annotated entity container, can contain dynamic properties as an effect of `$apply` even if they do not specify the `OpenType` attribute, see [OData-CSDL].

The term has a complex type with the following properties:

- **SupportsCount** indicates whether the virtual property can be used in.
- **The SupportedTransformations** collection lists all supported set transformations. Allowed values are the names of the standard transformations defined in chapter aggregate, topcount, tosum, 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 defined by this specification.
- **The CustomAggregationMethods collection lists all properties that can be used in the transformation.** If it is omitted the server supports all properties not listed as GroupableProperties (including the ) in the supported custom aggregation transformation.
- **The collection lists all dynamic properties that can be used in the transformation.**
- **The collection lists all properties that can be used in the 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 methods.** Allowed values are the AggregatableProperties and GroupableProperties namespace-qualified names identifying service-specific aggregation methods. If omitted, no custom aggregation methods are unspecified supported.
- **SupportsRollup** specifies to which extent whether the server supports no rollup, only a single rollup hierarchy, or multiple rollup hierarchies in a groupby transformation. If omitted, multiple rollup hierarchies are supported.
- **PropertyRestrictions specifies whether all properties can be used in groupby and aggregate.** If not specified, or specified with a value of false, all properties can be grouped and aggregated. If specified with a value of true clients have to check which properties are tagged as Groupable or Aggregatable.

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

**Example 35:** an entity container supporting everything defined in this specification.

```xml
<EntityContainer Name="SalesData">
  <Annotation Term="Aggregation.ApplySupported" />
  ...
</EntityContainer>
```
6.2 Property Annotations

6.2.1 Groupable Properties

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

6.2.2 Aggregatable Properties

A property that can be used in the transformation is described with the complex `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, and, or a namespace-qualified name identifying a provider-specific function, see section. 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 when using the aggregatable property in. 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 `is` annotated using the Measures vocabulary, see with `ApplySupported`).

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 or used within an additional collection property of the `SupportsApply` annotation on entity set or entity container that is annotated with `ApplySupported`, and the `ApplySupported` level. A dynamic aggregatable property is described by a complex type annotation has a value of `true` for `PropertyRestrictions`, only those properties that are annotated with the tagging term `Aggregatable` can be used in `aggregate` following properties:

6.2.3 Custom Aggregates

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

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

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

- The `Type` must be the value of the annotation is a string with the qualified name of a primitive type or type definition in scope, that specifies the type returned by the custom aggregate.
If the custom aggregate is associated with a structured type, the value of the Qualifier attribute MAY be identical to the name of a declared property of the structured type. In this case, the value of the Type property MUST have the same value as the Type attribute of the declared property. This is typically done when the custom aggregate is used as a default aggregate for that property. In this case, the name refers to the custom aggregate within an aggregate expression without a with clause, and to the property in all other cases.

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

Example 36: 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 custom aggregate of the Sales entity set type because it belongs there. For the budget, there is no appropriate entity set because there's no matching association between org units and time. Sales forecasts are modeled as a custom aggregate of the SalesData entity container level.

```xml
<Annotations Target="SalesModel.Container.Sales">
<Annotation Term="DataAggregation.SupportsApply">Aggregation.CustomAggregate" Qualifier="Forecast">
  <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>SalesOrganizations/Unit</PropertyPath>
            </Collection>
          </PropertyValue>
        </Record>
      </Collection>
    </PropertyValue>
  </Record>
  ...</Annotation>
</Annotations>

<Annotations Target="SalesModel.Container.SalesSalesData">
<Annotation Term="DataAggregation.SupportsApply">Aggregation.CustomAggregate" Qualifier="Budget">
  <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>
</Annotations>
```
These dynamic aggregatable properties custom aggregates can now be used in the aggregate transformation:

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

and:

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

### 6.31.1.1 Groupable Properties

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

- The Name contains the path to the groupable property.

### 6.2.4 The DependentProperties collection lists properties that are functionally dependent on the groupable property and should not be used in without this groupable property. In the , the customer Name could be listed Context-Defining Properties

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

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

### 6.2.5 Example

- **Example 37 a dependent:** This simplified Sales entity type has a single aggregatable property Amount whose context is defined by the Code property is used in the first parameter of without its groupable related Currency, and a custom aggregate Forecast with the same context. The Code property of Currency is groupable. All other properties are neither groupable nor aggregatable.
6.46.3 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**.

6.4.16.3.1 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**.

6.4.26.3.2 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.

### 6.4.2.16.3.2.1 Hierarchy Filter Functions

For testing the position of a given entity instance in a recursive hierarchy annotated to the entity’s type, the Aggregation vocabulary defines 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.

### 6.4.3.3 Examples

**Example 38: leveled hierarchies for products and time, and a recursive hierarchy for the sales organizations**

```xml
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx" Version="4.0">
    <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.saleservice">
      <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>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```
The recursive hierarchy SalesOrgHierarchy can be used in functions with the $filter system query option.

Example 39: requesting all organizations below EMEA

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

Example 40: requesting just those organizations directly below EMEA

```csharp
GET SalesOrganizations?$filter=it/Aggregation.isdescendant(Hierarchy='SalesOrgHierarchy', Node='EMEA',MaxDistance=1)
```
"@odata.context": "$metadata#SalesOrganizations",
"value": [
    { "ID": "EMEA Central", "Name": "EMEA Central" },
    { "ID": "EMEA South", "Name": "EMEA South" },
    { "ID": "EMEA North", "Name": "EMEA North" },
    ...
]
}

Example 41: just the lowest-level organizations

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


"@odata.context": "$metadata#SalesOrganizations",
"value": [
    { "ID": "Sales Office London", "Name": "Sales Office London" },
    { "ID": "Sales Office New York", "Name": "Sales Office New York" },
    ...
]
}

Example 42: the lowest-level organizations including their superordinate’s ID

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

"@odata.context": "$metadata#SalesOrganizations(*,Superordinate(ID))",
"value": [
    { "ID": "Sales Office London", "Name": "Sales Office London", "Superordinate: { "ID": "EMEA United Kingdom" } },
    ...
]
}

Example 43: retrieving the sales IDs involving sales organizations from EMEA can be requested by

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

{ "$odata.context": "$metadata#Sales(ID)",
**6.56.4 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 44:* 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 45:* 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

```json
{
    "@odata.context": "http://host/service/$metadata#Sales(<properties in $apply>)",
    "@odata.readLink": "http://.../aggregated-stuff2143248437259843",
    "#Model.ColAction": {
        "target": "http://.../aggregated-stuff2143248437259843/Model.ColAction"
    },
    "value": [
        {
            "@odata.id": "aggregated-stuff2143248437259843-1",
            "#Model.SingleAction": {
                "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

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

Example 46:

```
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 47: 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 48:

```
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" },
    "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" }
  }
]

This could also have been formulated as

```plaintext
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 49: the first question in the motivating example in section 2.4, which customers bought which products, can now be expressed as*

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

and results in

```plaintext
{
  "@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 Methods

The client may specify one of the predefined aggregation methods `min`, `max`, `sum`, `average`, and `countdistinct`, or a custom aggregation method, to aggregate an aggregatable property. Expressions defining an aggregate method MUST specify an alias. The aggregated values will be returned in the entity's dynamic property whose values are aggregated, of course without changing its type; the name is determined by the alias.

*Example 50:*
GET ~/Products?$apply=groupby((Name),
 aggregate(Sales/Amount with sum as Total))

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 item. If there are no “base” entities to be aggregated, the array is empty.

Example 51: 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 with sum as Total))

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 52:

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

returns the different products sold per country:
Example 53:

### 7.3 Consumer-Controlled Aggregation

Instead of using the provider-defined default aggregation behavior, the consumer may specify one of the predefined aggregation methods, and:

**Example:**

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

results in

```
{ "@odata.context": "$metadata#Sales(Country,TotalAmount)",
  "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" } } ]
  ]
}
```

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:**

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

results in

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

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 54:

```
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 property has the same navigation path as the original value.

Example 55: the result properties for Sales$/count and Sales/Amount are placed differently

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

results in

```
{ "@odata.context": "$metadata#Products(Name,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": null } ] },
    { "@odata.id": null, "Name": "Sugar", "SalesCount": 2,
      "Sales": [ { "TotalAmount": 4 } ] }
  ]
}
```

To place the number of instances in a group next to other aggregated values, the virtual property $count can be used:

Example 56: the result properties for Sales$/count and Sales/Amount are placed differently

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

results in

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

Custom aggregates are defined through the `CustomAggregate` annotation. They can be associated with either an entity type or an entity container.

A custom aggregate can be used by specifying the name of the custom aggregate in the `aggregate` clause.

**Example 57:**

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

Results in

```
{ "@odata.context": "$metadata#Sales(Customer(Country),Actual,Forecast))",
  "value": [
    { "@odata.id": null, "Customer": { "Country": "Netherlands" },
      "Actual": 5, "Forecast": 4 },
    { "@odata.id": null, "Customer": { "Country": "USA" },
      "Actual": 19, "Forecast": 21 }
  ]
}
```

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

**Example 58:**

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

Results in

```
[ { "@odata.context": "$metadata#Products(Name,Sales(Actual,Forecast))",
  "value": [ { "@odata.id":null,"Name":"Coffee","Sales":{"Actual":12,"Forecast":6} },
              { "@odata.id":null,"Name":"Paper","Sales":{"Actual":8,"Forecast":2} },
              { "@odata.id":null,"Name":"Pencil","Sales":[] },
              { "@odata.id":null,"Name":"Sugar","Sales":{"Actual":4,"Forecast":7} } ]
]
```

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

**Example 59** A custom aggregate can be defined with the same name as a property of the same type in order to define a default aggregate for that property.

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

Results in

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

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

Example 60:

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

```
results in
|
```

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

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

Example 53:

```
Example 61:
```

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

```
results in
|
```

```
+ "@odata.context": "$metadata#Products(Name,Sales(AmountTotal,AvgAmt))",
  "value": [
```

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

Example 62:

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

will return all distinct amounts appearing in sales orders and how much money was made with deals of this amount.
7.5 Combining Transformations per Group

Example 63: to get the best-selling product per country with sub-totals for every country, the partial results of a transformation sequence and a groupby transformation are concatenated:

```
GET ~/Sales?$apply=concat(
    groupby((Customer/Country,Product/Name,Currency/Code),
        aggregate(Amount with sum as Total))
    /groupby((Customer/Country,Currency/Code),
        topcount(1,AmountTotal)),
    groupby((Customer/Country,Currency/Code),
        aggregate(Amount with sum as Total)))
```

Example 64: transformation sequences are also useful inside groupby: To get the aggregated amount by only considering the top two sales amounts per product and county:

```
GET ~/Sales?$apply=groupby((Customer/Country,Product/Name,Currency/Code),
    topcount(2,Amount)/aggregate(Amount with sum as Total))
```
7.6 Applying Model Functions as Set Transformations

**Example 65**: 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 a parameter**, retains those entities that `topcount` also would retain, and replaces the remaining entities by a single aggregated entity, where only the numeric property has a defined value being the aggregated value over those remaining entities–:

**The request**

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

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

```"@odata.context":
"$metadata#Sales(Customer(Country),Product(Name),AmountTotal,Currency(Code))","value": [  
  { "@odata.id": null, "Customer": { "Country": "Netherlands" },  
    "Product": { "Name": "Paper" },  
    "AmountTotal": 3, "Currency": { "Code": "EUR" }  
  },  
  { "@odata.id": null, "Customer": { "Country": "Netherlands" },  
    "Product": { "Name": "Sugar" },  
    "AmountTotal": 2, "Currency": { "Code": "EUR" }  
  },  
  { "@odata.id": null, "Customer": { "Country": "USA" },  
    "Product": { "Name": "Coffee" },  
    "AmountTotal": 12, "Currency": { "Code": "USD" }  
  },  
  { "@odata.id": null, "Customer": { "Country": "USA" },  
    "Product": { "Name": "Paper" },  
    "AmountTotal": 5, "Currency": { "Code": "USD" }  
}]
```
"Product": { "Name": "Paper" }, "AmountTotal": 3 },
{ "@odata.id": null, "Customer": { "Country": "Netherlands" },
 "Product": { "Name": "**Other**" }, "AmountTotal": 2 },
{ "@odata.id": null, "Customer": { "Country": "USA" },
 "Product": { "Name": "Coffee" }, "AmountTotal": 12 },
{ "@odata.id": null, "Customer": { "Country": "USA" },
 "Product": { "Name": "**Other**" }, "AmountTotal": 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 66:** get the average of the overall amount by month per product.

Using a transformation sequence:

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

Using `from`:

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

**Example 67:** for an aggregate entity set listing the total sales amounts per customer and country, the rollup shall produce additional entity instances for the average total sales amount of customers per country and the average of that average (which is a bit boring because the example data doesn’t have two countries with the same currency 😅)

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

results in

```plaintext
{
    "$@odata.context":
    "$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,
    "Customer": {
      "Country": "Netherlands",
      "ID": "C3"
    },
    "CustomerCountryAverage": 9.5,
    "Currency": {
      "Code": "USD"
    }
  },
  {
    "@odata.id": null,
    "Customer": {
      "Country": "Netherlands"
    },
    "CustomerCountryAverage": 5,
    "Currency": {
      "Code": "EUR"
    }
  }
]

7.8 Transformation Sequences

Applying aggregation first covers the most prominent use cases, yet it is insufficient to answer. The slightly 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 68:

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

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

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

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

Example 69:

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

&$filter=AmountTotal ge 4

results in

```
{
    "$odata.context": "$metadata#Sales(Product(Name),AmountTotal)",
    "value": [
        {
            "$odata.id": null,
            "AmountTotal": 4,
            "Product": { "Name": "Paper" }
        },
        {
            "$odata.id": null,
            "AmountTotal": 4,
            "Product": { "Name": "Sugar" }
        }
    ]
}
```
**Example 70:** Revisiting the Example 13 in section 3.1.4 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 with sum as Total))
/aggregate(Amount as Total with average as DailyAverage)
```

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

**Example 71:** getting the population per country with

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

```
results in

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

**Example 72:** all countries with megacities and their continents

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

**Example 73:** all countries with tens of millions of city dwellers and the continents only for these countries

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

**Example 74:** all countries with tens of millions of city dwellers and all continents with cities independent of their size

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

Example 75: filter the base set and filter related items before aggregation

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

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Editor</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working Draft 01</td>
<td>2012-11-12</td>
<td>Ralf Handl</td>
<td>Translated contribution into OASIS format</td>
</tr>
<tr>
<td>Committee Specification Draft 01</td>
<td>2013-07-25</td>
<td>Ralf Handl, Hubert Heijkers, Gerald Krause, Michael Pizzo, Martin Zurmuehl</td>
<td>Switched to pipe-and-filter-style query language based on composable set transformations</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fleshed out examples and addressed numerous editorial and technical issues processed through the TC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Added Conformance section</td>
</tr>
<tr>
<td>Committee Specification Draft 02</td>
<td>2014-01-09</td>
<td>Ralf Handl, Hubert Heijkers, Gerald Krause, Michael Pizzo, Martin Zurmuehl</td>
<td>Dynamic properties used all aggregated values, either via aliases or via custom aggregates</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Refactored annotations</td>
</tr>
</tbody>
</table>