OData Extension for Data Aggregation Version 4.0

Committee Specification ~~02~~

Draft 04 ~~November 2015~~

~~Specification URIs~~

05 July 2023

This ~~version~~stage:

https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csd04/odata-data-aggregation-ext-v4.0-csd04.md (Authoritative)
https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csd04/odata-data-aggregation-ext-v4.0-csd04.html
https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csd04/odata-data-aggregation-ext-v4.0-csd04.pdf

Previous stage:

https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/cs02/odata-data-aggregation-ext-v4.0-cs02.docx (Authoritative)
https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/cs02/odata-data-aggregation-ext-v4.0-cs02.html
https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/cs02/odata-data-aggregation-ext-v4.0-cs02.pdf

~~Previous version:~~

Latest stage:

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

~~Latest version:~~
~~(Authoritative)~~

Technical Committee:

OASIS Open Data Protocol (OData) TC

Chairs:

Ralf Handl (ralf.handl@sap.com), SAP SE

~~Ram Jeyaraman~~
Michael Pizzo (mikep@microsoft.com), Microsoft

Editors:

Ralf Handl (ralf.handl@sap.com), SAP SE
Hubert Heijkers (hubert.heijkers@nl.ibm.com), IBM
Gerald Krause (gerald.krause@sap.com), SAP SE
Michael Pizzo (mikep@microsoft.com), Microsoft
Heiko Theißen (heiko.theissen@sap.com), SAP SE
Martin Zurmühl (martin.zurmuehl@sap.com), SAP SE

~~Gerald Krause (),~~
~~Michael Pizzo (),~~

~~Martin Zurmuehl (),~~ Additional artifacts:

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

- ABNF components: *OData Aggregation ABNF Construction Rules Version 4.0*~~.~~ and *OData Aggregation ABNF Test Cases*: https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csd04/abnf/
- OData Aggregation Vocabulary:

    - https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csd04/vocabularies/Org.OData.Aggregation.V1.json
    - https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csd04/vocabularies/Org.OData.Aggregation.V1.xml

Related work:

This specification is related to:

- *OData Version 4.~~0~~01*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. A multi-part Work Product ~~that~~which includes:
    - *OData Version 4.~~0~~01 Part 1: Protocol*. Latest ~~version.~~stage: https://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part1-protocol.html
    - *OData Version 4.~~0~~01 Part 2: URL Conventions*. Latest ~~version.~~stage: https://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part2-url-conventions.html
    - ~~*OData Version 4.0 Part 3: Common Schema Definition Language (CSDL)*. Latest version.~~ *ABNF components: OData ABNF Construction Rules Version 4.*~~*0*. 30 October 2014.~~ *01 and OData ABNF Test Cases*. ~~30 October 2014.~~ https://docs.oasis-open.org/odata/odata/v4.~~0/errata02~~01/os~~/complete~~/abnf/
- *OData Vocabularies Version 4.0*. Edited by Michael Pizzo, Ralf Handl, and Ram Jeyaraman. Latest stage: https://docs.oasis-open.org/odata/odata-vocabularies/v4.0/odata-vocabularies-v4.0.html
    - *OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. Latest stage: https://docs.oasis-open.org/odata/odata-csdl-json/v4.01/odata-csdl-json-v4.01.html~~*OData Core Vocabulary*. 30 October 2014.~~
- ~~*OData Measures Vocabulary*. 30 October 2014.~~
- *OData Common Schema Definition Language (CSDL) XML Representation Version 4.01*. Edited by Michael Pizzo, Ralf Handl, and Martin Zurmuehl. Latest stage: https://docs.oasis-open.org/odata/odata-csdl-xml/v4.01/odata-csdl-xml-v4.01.html
- *OData JSON Format Version 4.~~0~~01*. Edited by Ralf Handl, Mike Pizzo, and Mark Biamonte. Latest ~~version.~~stage: https://docs.oasis-open.org/odata/odata-json-format/v4.01/odata-json-format-v4.01.html

Abstract:

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

Status:

This document was last revised or approved by the OASIS Open Data Protocol (OData) TC on the above date. The level of approval is also listed above. Check the ~~“~~"Latest ~~version”~~stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata#technical.

TC members should send comments on this specification to the TC~~’~~'s email list. Others should send comments to the TC~~’~~'s public comment list, after subscribing to it by following the instructions at the "Send A Comment~~“”~~" button on the TC~~’~~'s web page at https://www.oasis-open.org/committees/odata/.

This specification is provided under the RF on RAND Terms Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. 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 TC~~’~~'s web page (https://www.oasis-open.org/committees/odata/ipr.php).

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Key words:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] and [RFC8174] when, and only when, they appear in all capitals, as shown here.

Citation format:

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

[OData-Data-Agg-v4.0]

OData Extension for Data Aggregation Version 4.0. Edited by Ralf Handl, Hubert Heijkers, Gerald Krause, Michael Pizzo, Heiko Theißen, and Martin Zurmuehl. ~~04 November 2015.~~05 July 2023. OASIS Committee Specification ~~02~~Draft 04. https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/csd04/odata-data-aggregation-ext-v4.0-csd04.html. Latest ~~version~~stage: https://docs.oasis-open.org/odata/odata-data-aggregation-ext/v4.0/odata-data-aggregation-ext-v4.0.html.

Notices

For complete copyright information please see the full Notices section in an Appendix below~~Please see for above guidance.~~

.

Table of Contents

~~1 Introduction~~

1 Introduction

This specification adds ~~the notion of~~ aggregation functionality 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 Glossary

1.1.1 Definitions of Terms

~~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 **[]**.~~

~~1.2 Normative References~~

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

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

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

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

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

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

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

~~**[OData-VocMeas]** *OData Measures Vocabulary*.~~
~~See link in "Related work" section on cover page.~~

~~**[RFC2119]** *Bradner, S.,* "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997. .~~

~~1.3 Non-Normative References~~

~~[TSQL ROLLUP]~~

~~1.4 Typographical Conventions~~

This specification defines the following terms:

- *Aggregatable Expression* – an expression not involving term casts and resulting in a value of a complex or entity or an aggregatable primitive type
- *Aggregate Expression* – argument of the aggregate transformation or function defined in section 3.2.1.1
- *Aggregatable Primitive Type* – a primitive type other than Edm.Stream or subtypes of Edm.Geography or Edm.Geometry
- *Data Aggregation Path* – a path that consists of one or more segments joined together by forward slashes (/). Segments are names of declared or dynamic structural or navigation properties, or type-cast segments consisting of the (optionally qualified) name of a structured type that is derived from the type identified by the preceding path segment to reach properties declared by the derived type.
- *Expression* – derived from the commonExpr rule (see OData-ABNF)
- *Single-Valued Property Path* – property path ending in a single-valued primitive, complex, or navigation property

1.1.2 Acronyms and Abbreviations

The following non-exhaustive list contains variable names that are used throughout this document:

- \(A,B,C\) – collections of instances
- \(H\) – hierarchical collection
- \(H'\) – subset of nodes from a hierarchical collection
- \(u,v,w\) – instances in a collection
- \(x\) – an instance in a hierarchical collection, called a node
- \(p,q,r\) – paths
- \(S,T\) – transformation sequences
- \(α\) – aggregate expression, defined in section 3.2.1.1
- \(\Gamma(A,p)\) – the collection that results from evaluating a data aggregation path \(p\) relative to a collection \(A\), defined in section 3.1.3

- \(γ(u,p)\) – the collection that results from evaluating a data aggregation path \(p\) relative to an instance \(u\), defined in section 3.1.3
- \(\Pi_G(s)\) – a transformation of a collection that injects grouping properties into every instance of the collection, defined in section 3.2.3.1
- \(σ(x)\) – instance containing a grouping property that represents a node \(x\), defined in section 6.2.2

1.1.3 Document 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. Examples labeled with ⚠ contain advanced concepts or make use of keywords that are defined only later in the text, they can be skipped at first reading.

All other text is normative unless otherwise labeled.

Here is a customized command line which will generate HTML from this markdown file (named odata-data-aggregation-ext.md). Line breaks are added for readability only:

```
pandoc -f gfm+tex_math_dollars+fenced_divs
       -t html
       -o odata-data-aggregation-ext.html
       -c styles/markdown-styles-v1.7.3b.css
       -c styles/odata.css
       -s
       --mathjax
       --eol=lf
       --wrap=none
       --metadata pagetitle="OData Extension for Data Aggregation Version 4.0"
       odata-data-aggregation-ext.md
```

~~2~~ This uses pandoc 3.1.2 from https://github.com/jgm/pandoc/releases/tag/3.1.2~~Overview~~

.

2 Overview

Open Data Protocol (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~~aggregation annotations 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 express the desired aggregation behavior for a particular query. They also allow consumers to formulate queries that ~~refer to~~utilize the aggregation annotations~~ as shorthand~~.

~~2.1 Definitions~~

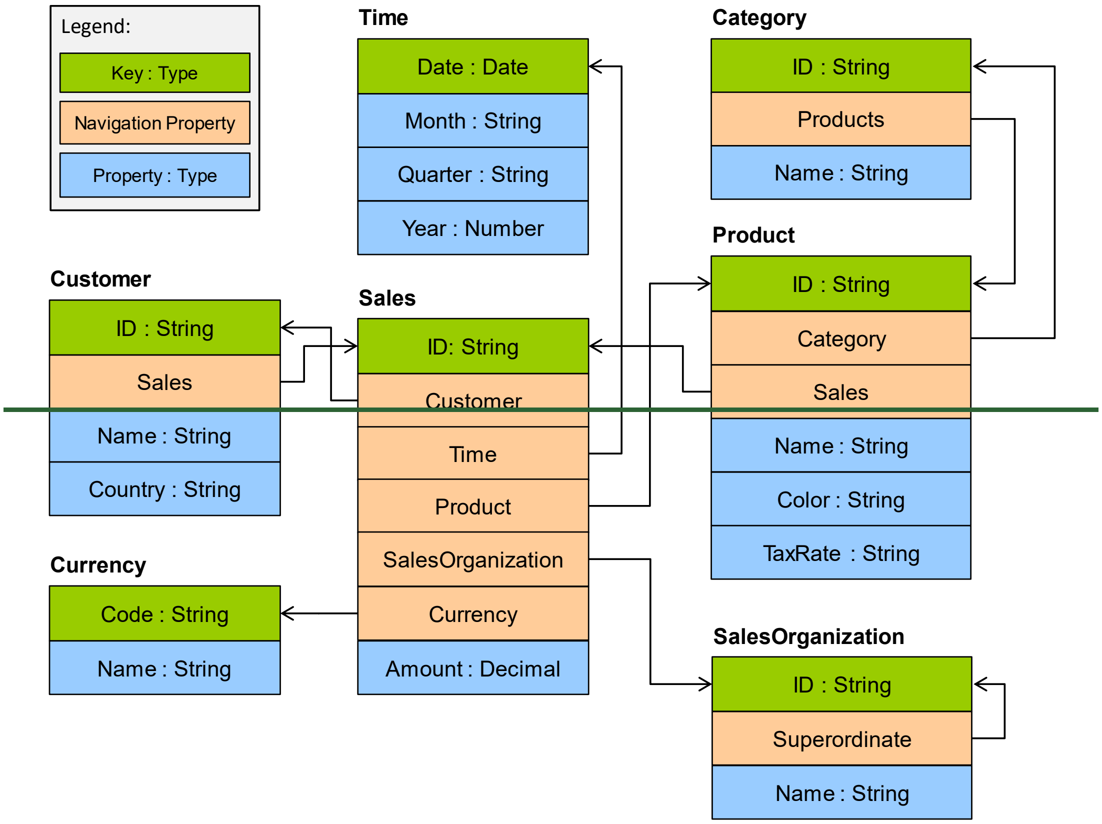
~~This specification defines the following terms:~~

2.1 Example Data Model

- ~~– a property for which the values can be aggregated using an aggregation method.~~
- ~~– a method that can be used to aggregate an aggregatable property or expression~~
- ~~*Standard Aggregation Method* – one of the standard aggregation methods: , , , , and~~
- ~~– a custom aggregation method that can be applied to expressions of a specified type~~
- ~~– a dynamic property that can appear in an aggregate clause~~
- ~~– a property whose values can be used to group entities or complex type instances for aggregation.~~
- ~~– 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~~depicts a simple model that is used throughout this document.



ID: Edm.String {id} Amount: Edm.Decimal Sale Date: Edm.Date {id} Month: Edm.String Quarter: Edm.String Year: Edm.Int16 Time ID: Edm.String {id} Name: Edm.String Country: Edm.String Customer ID: Edm.String {id} Name: Edm.String Category ID: Edm.String {id} Name: Edm.String Color: Edm.String TaxRate: Edm.Decimal Product ID: Edm.String {id} Name: Edm.String SalesOrganization 1 * 1 * * 1 0..1 * 1 * 1 * Sales Sales

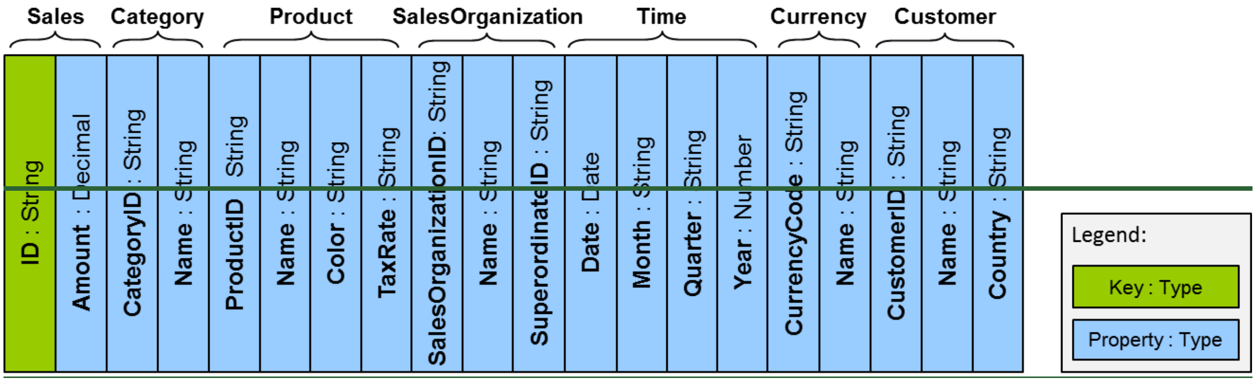
Customer Product Products Category Time SalesOrganization Superordinate Rating: Edm.Byte FoodProduct RatingClass: Edm.String NonFoodProduct

The `Amount` property in the `Sale~~s~~` entity type is an aggregatable property~~aggregatable property,~~, and the properties of the related entity types are groupable. These can be arranged in ~~four~~ hierarchies, for example:

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

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

Query extensions and descriptive annotations can ~~both~~ be applied to normalized schemas 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 diagram depicts a denormalized schema for the simple model.*

Sale	
Sales	ID: Edm.String {id}
	Amount: Edm.Decimal
Category	CategoryID: Edm.String

	CategoryName: Edm.String
Product	ProductID: Edm.String
	ProductName: Edm.String
	ProductColor: Edm.String
	ProductTaxRate: Edm.Decimal
Food	FoodProductRating: Edm.Byte
Non-Food	NonFoodProductRatingClass: Edm.String
Sales Organization	SalesOrganizationID: Edm.String
	SalesOrganizationName: Edm.String
	SalesOrganizationSuperordinateID: Edm.String
Time	TimeDate: Edm.Date
	TimeMonth: Edm.String
	TimeQuarter: Edm.String
	TimeYear: Edm.Int16
Customer	CustomerID: Edm.String
	CustomerName: Edm.String
	CustomerCountry: Edm.String

2.2 Example Data

Example 4: The following entity sets and sample data will be used to further illustrate the capabilities introduced by this extension.

Products

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

ID	**Category**	**Name**	**Color**	**TaxRate**
P4	PG2	Pencil	Black	0.14

Food

Rating
5
n/a
n/a

Non-Food

RatingClass
n/a
n/a
average

Time

Date	**Month**	**Quarter**	**Year**
2022-01-01	2022-01	2022-1	2022
2022-04-01	2022-04	2022-2	2022
2022-04-10	2022-04	2022-2	2022
...			

Categories

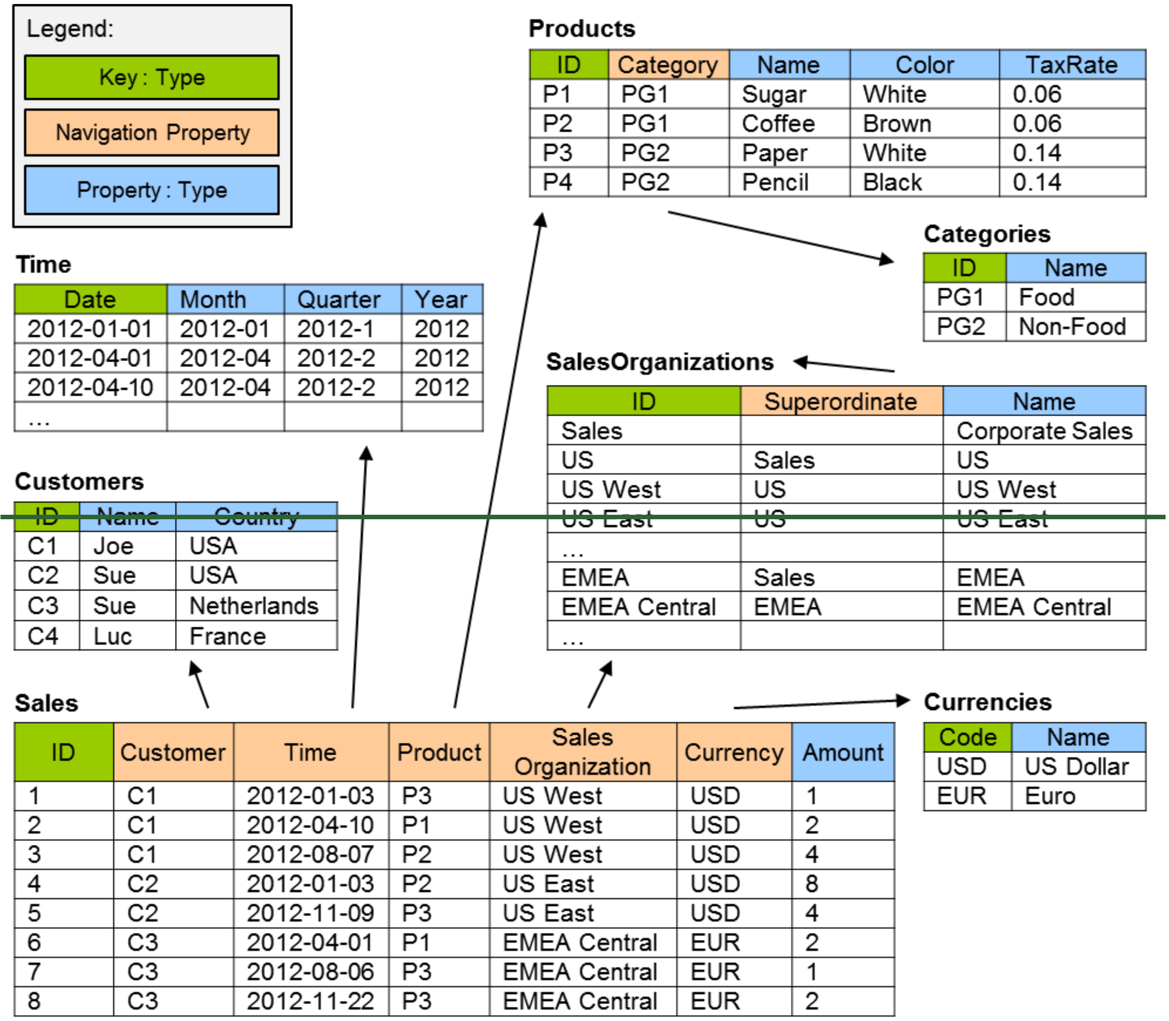
ID	Name
PG1	Food
PG2	Non-Food

Sales Organizations

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

Customers

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



Products

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

Categories

ID	Name
PG1	Food
PG2	Non-Food

Time

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

SalesOrganizations

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

Customers

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

Sales

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

Currencies

Code	Name
USD	US Dollar
EUR	Euro

Sales

~~2.4 Example Use Cases~~

ID	Customer	Time	Product	Sales Organization	Amount
1	C1	2022-01-03	P3	US West	1
2	C1	2022-04-10	P1	US West	2
3	C1	2022-08-07	P2	US West	4
4	C2	2022-01-03	P2	US East	8
5	C2	2022-11-09	P3	US East	4
6	C3	2022-04-01	P1	EMEA Central	2

ID	Customer	Time	Product	Sales Organization	Amount
7	C3	2022-08-06	P3	EMEA Central	1
8	C3	2022-11-22	P1	EMEA Central	2

Legend

Property
Key
Navigation Property

2.3 Example Use Cases

Example ~~:~~5: In the example model, one prominent use case is the relation of customers to products. The first question that is likely to be asked is: ~~“~~"Which customers bought which products~~?”~~ ?"

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

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

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

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

Customer/Country	Customer/Name	Product/Category/Name	Product/Name	Amount	~~Currency/Code~~
USA	Joe	Non-Food	Paper	1	~~USD~~

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

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

~~3~~ 3 System Query Option $apply~~System Query Option $apply~~

~~Aggregation behavior~~

A *set transformation* (*transformation* for short) is ~~triggered using the query option $apply. It takes~~an operation on an input set that produces an output set. A *transformation sequence* is a sequence of set transformations, separated by forward slashes to express that they are consecutively applied~~, i.e. the result of each~~. A transformation sequence may be invoked using the system query option $apply. The input set of the first set transformation is the ~~input to the next transformation.~~collection addressed by the resource path. The output set of each set transformation is the input set for the next set transformation. The output set of the last set transformation in the transformation sequence invoked by the system query option $apply is the result of $apply. This is consistent with the use of service-defined ~~bindable~~bound and composable functions in path segments. Set transformations may also appear as a parameter of certain other set transformations defined below.

~~Unless otherwise noted, each set transformation:~~

~~preserves the structure of the input type, so the structure of~~The system query option $apply MUST NOT be used if the resource path addresses a single instance.

The system query option $apply is evaluated first, then the other system query options are evaluated, if applicable, on the result of $apply, see OData-Protocol, section 11.2.1. Stability across requests for system query options $top and $skip OData-Protocol, sections 11.2.6.3 and 11.2.6.4 is defined in section 3.3.7.

Each set transformation:

- carries over the input type to the output set such that it fits into the data model of the service.
- ~~does not necessarily preserve the~~can mark certain navigation properties and stream properties for *expansion by default*, that is, they are expanded in the result of $apply in the absence of an $expand query option.
- may produce an output set with a different number of instances ~~in the result, as this will typically differ from the number of instances in~~than the input set.

- does not necessarily guarantee that all properties of the ~~result~~ instances in the output set have a well-defined value.

Instances of an output set can contain structural and navigation properties, which can be declared or dynamic, as well as instance annotations.

The allowed set transformations are defined in this section as well as in the section on Hierarchical Transformations~~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 **[]**, with dynamic properties representing the aggregate values. The parameters of set transformations allow specifying how the result instances are constructed from the input instances.~~
~~The set transformations defined by this extension are~~

.

Service-defined bound functions that take ~~an entity set~~ a collection of instances of a structured type as their binding parameter and return a collection of instances of a structured type 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~~. 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 7.7~~section~~ 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.

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

During serialization of the result of $apply declared properties and dynamic properties are represented as defined by the response format. Other properties have been aggregated away and are not represented in the response. The entities returned in the request examples in the following sections that involve aggregation are therefore transient.

3.1 Fundamentals of Input and Output Sets

The definitions of italicized terms made in this section are used throughout this text, always with a hyperlink to this section.

3.1.1 Type, Structure and Context URL

All input sets and output sets in one transformation sequence are collections of the *input type*, that is the entity type or complex type of the first input set, or in other words, of the resource to which the transformation sequence is applied. The input type is determined by the entity model element identified within the metadata document by the context URL of that resource OData-Protocol, section 10. Individual instances in an input or output set can have a subtype of the input type. (See example 74.) The transformation sequence given as the $apply system query option is applied to the resource addressed by the resource path. The transformations defined below can have nested transformation sequences as parameters, these are then applied to resources that can differ from the current input set.

The *structure* of an instance that occurs in an input or output set is defined by the names of the structural and navigation properties that the instance contains. Instances of an input type can have different structures, subject to the following rules:

- Declared properties of the input type or a nested or related type thereof or of a subtype of one of these MUST have their declared type and meaning when they occur in an input or output set.
- Single- or collection-valued primitive properties addressed by a property path starting at a non-transient entity MUST keep their values from the addressed resource path collection throughout the transformation sequence. Likewise, single- or collection-valued navigation property paths starting at a non-transient entity MUST keep addressing the same non-transient entities as in the addressed resource path collection.
- Instances in an output set need not have all declared or dynamic properties that occurred in the input set.
- Instances in an output set can have dynamic properties that did not occur in the input set. The name for such a dynamic property is called an *alias*, it is a simple identifier (see OData-CSDL, section 17.2). Aliases MUST differ from names of declared properties in the input type, from names of properties in the first input set, and from names of properties in the current input set. Aliases in one collection MUST also differ from each other.

Here is an overview of the structural changes made by different transformations:

- During aggregation or nest, many instances are replaced by one instance, properties that represent the aggregation level are retained, and others are replaced by dynamic properties holding the aggregate value of the many instances or a transformed copy of them.

- During compute, dynamic properties are added to each instance.
- During addnested, dynamic properties are added to each occurrence of a related collection.
- During join, one instance with a collection of related instances is replaced by many copies, each of which is related via a dynamic property to one of the related instances.
- During concatenation, the same instances are transformed multiple times and the output sets with their potentially different structures are concatenated.

An output set thus consists of instances with different structures. This is the same situation as with a collection of an open type OData-CSDL, sections 6.3 and 9.3 and it is handled in the same way.

If the first input set is a collection of entities from a given entity set, then so are all input sets and output sets in the transformation sequence. The {select-list} in the context URL OData-Protocol, section 10 MUST describe only properties that are present or annotated as absent (for example, if Core.Permissions is None OData-Protocol, section 11.2.2) in all instances of the collection, after applying any $select and $expand system query options. The {select-list} SHOULD describe as many such properties as possible, even if the request involves a concatenation that leads to a non-homogeneous structure. If the server cannot determine any such properties, the {select-list} MUST consist of just the instance annotation AnyStructure defined in the Core vocabulary OData-VocCore. (See example 75.)

3.1.2 Sameness and Order

Input sets and output sets are not sets of instances in the mathematical sense but collections, because the same instance can occur multiple times in them. In other words: A collection contains values (which can be instances of structured types or primitive values), possibly with repetitions. The occurrences of the values in the collection form a set in the mathematical sense. The *cardinality* of a collection is the total number of occurrences in it. When this text describes a transformation algorithmically and stipulates that certain steps are carried out *for each occurrence* in a collection, this means that the steps are carried out multiple times for the same value if it occurs multiple times in the collection.

A collection addressed by the resource path is returned by the service either as an ordered collection OData-Protocol, section 11.4.10 or as an unordered collection. The same applies to collections that are nested in or related to the addressed resource as well as to collections that are the result of evaluating an expression starting with $root, which occur, for example, as the first parameter of a hierarchical transformation.

But when such a collection is transformed by the $apply system query option, additional cases can arise that are neither ordered nor totally unordered. For example, the groupby transformation retains any order within a group but not between groups.

⚠ Example 6: Request the top 10 sales per customer. The processing of the request can be parallelized per customer and the responses per customer can be interleaved in the overall response. This means that for any given customer, their top 10 sales appear in the desired order, though not consecutively.

```
GET /service/Sales?$apply=groupby((Customer),orderby(Amount
desc)/top(10))
```

For every transformation defined in the following sections, it will be specified how it orders its output set, based on the order of its input set. The order of the last output set can be further influenced by a $orderby system query option before it is observed in the response payload.

An order of a collection is more precisely defined as follows: Given two different occurrences u_1 and u_2 in a collection, which may be of the same value or of different values, u_1 precedes u_2 or u_2 precedes u_1, but not both. It can be neither, in which case the relative order of u_1 and u_2 does not matter. If u_1 precedes u_2 and u_2 precedes u_3, then u_1 also precedes u_3, and u_1 never precedes u_1. (This is a partial order in the mathematical sense defined on the set of occurrences.)

When transformations are defined in the following sections, the algorithmic description sometimes contains an *order-preserving loop* over a collection. Such a loop processes the occurrences in an order chosen by the service in such a way that u_1 is processed before u_2 whenever u_1 precedes u_2. Likewise, in an *order-preserving sequence* $u_1,…,u_n$ we have $i<j$ whenever u_i precedes u_j.

A collection can be *stable-sorted* by a list of expressions. In the stable-sorted collection an occurrence u_1 precedes u_2 if and only if either

- u_1 precedes u_2 according to the rules of OData-Protocol, section 11.2.6.2 or
- these rules do not determine a precedence in either direction between u_1 and u_2 but u_1 preceded u_2 in the collection before the sort.

Stable-sorting of an ordered collection produces another ordered collection. A stable-sort does not necessarily produce a total order, the sorted collection may still contain two occurrences whose relative order does not matter. The transformation `orderby` performs a stable-sort.

The output set of a basic aggregation transformation can contain instances of an entity type without entity id. After a `concat` transformation, different occurrences of the same entity can differ in individual non-declared properties. To account for such cases, the definition of sameness given in OData-URL, section 5.1.1.1.1 is refined here. Instances of structured types are *the same* if

- both are instances of complex types and both are null or both have the same structure and same values with null considered different from absent or
- both are instances of entity types without entity id (transient entities, see OData-Protocol, section 4.3) and both are null or both have the same structure and same values with null considered different from absent (informally speaking, they are compared like complex instances) or
- (1) both are instances of the same entity type with the same entity id (non-transient entities, see OData-Protocol, section 4.1) and (2) the structural and navigation properties contained in both have the same values (for non-primitive properties the sameness of values is decided by a recursive invocation of this definition).
  - If this is fulfilled, the instances are called *complementary representations of the same non-transient entity*. If this case is encountered at some recursion level while the sameness of non-transient entities \(u_1\) and \(u_2\) is established, a merged representation of the entity \(u_1=u_2\) exists that contains all properties of \(u_1\) and \(u_2\). But if the instances both occur in the last output set, services MUST represent each with its own structure in the response payload.
  - If the first condition is fulfilled but not the second, the instances are not the same and are called *contradictory representations of the same non-transient entity*. (Example 103 describes a use case for this.)

Collections are *the same* if there is a one-to-one correspondence \(f\) between them such that

- corresponding occurrences are of the same value and
- an occurrence \(u_1\) precedes another occurrence \(u_2\) if and only if the occurrence \(f(u_1)\) precedes the occurrence \(f(u_2)\), where the occurrences \(u_1\) and \(u_2\) may be of the same

value or of different values. (A one-to-one correspondence with this second property is called *order-preserving*.)

3.1.3 Evaluation of Data Aggregation Paths

This document specifies how a data aggregation path that occurs in a request is evaluated by the service. If such an evaluation fails, the service MUST reject the request.

For a data aggregation path to be a common expression according to OData-URL, section 5.1.1, its segments must be single-valued with the possible exception of the last segment, and it can then be evaluated relative to an instance of a structured type. For the transformations defined in this document, a data aggregation path can also be evaluated relative to a collection \(A\), even if it has arbitrary collection-valued segments itself.

To this end, the following notation is used in the subsequent sections: If \(A\) is a collection and \(p\) a data aggregation path, optionally followed by a type-cast segment, the result of such a path evaluation is denoted by \(\Gamma(A,p)\) and defined as the unordered concatenation, possibly containing repetitions, of the collections \(γ(u,p)\) for each \(u\) in \(A\) that is not null. The function \(γ(u,p)\) takes a non-null value and a path as arguments and returns a collection of instances of structured types or primitive values, depending on the type of the final segment of \(p\). It is recursively defined as follows:

1. If \(p\) is an empty path, let \(B\) be a collection with \(u\) as its single member and continue with step 9.
2. Let \(p_1\) be the first segment of \(p\) and \(p_2\) the remainder, if any, such that \(p\) equals the concatenated path \(p_1/p_2\).
3. If \(p_1\) is a type-cast segment and \(u\) is of its type or a subtype thereof, let \(v=u\) and continue with step 8.

~~3.1~~ If \(p_1\) is a type-cast segment and \(u\) is not of its type or a subtype thereof, let \(B\) be an empty collection and continue with step 9. (This rule follows OData-URL, section 4.11 rather than OData-CSDL, section 14.4.1.1~~Transformation aggregate~~

4. .)
5. Otherwise, \(p_1\) is a non-type-cast segment. If \(u\) does not contain a structural or navigation property \(p_1\), let \(B\) be an empty collection and continue with step 9.
6. If \(p_1\) is single-valued, let \(v\) be the value of the structural or navigation property \(p_1\) in \(u\). If \(v\) is null, let \(B\) be an

empty collection and continue with step 9; otherwise continue with step 8.

7. Otherwise, \(p_1\) is collection-valued. Let \(C\) be the collection addressed by the structural or navigation property \(p_1\) in \(u\), and let \(B=\Gamma(C,p_2)\). Then continue with step 9.
8. Let \(B=γ(v,p_2)\).
9. Return \(B\).

This notation is extended to the case of an empty path \(e\) by setting \(\Gamma(A,e)=A\) with null values removed. Note the collections returned by \(\Gamma\) and \(γ\) never contain the null value. Also, every instance \(u\) in \(\Gamma(A,p)\) occurs also in \(A\) or nested into \(A\), therefore an algorithmic step like "Add a dynamic property to each \(u\) in \(\Gamma(A,p)\)" effectively changes \(A\).

3.2 Basic Aggregation

3.2.1 Transformation aggregate

3.2.1.1 Aggregation Algorithm

The aggregate transformation takes a comma-separated list of one or more *aggregate expressions*~~*aggregate expressions*~~ as parameters and returns ~~a result~~an output set with a single instance of the input type without entity id containing one property per aggregate expression, representing the aggregated value ~~for all instances in~~of the input set.

An aggregate expression MUST have one of the types listed below or be constructed with the from keyword. To compute the value of the property for a given aggregate expression, the aggregate transformation first determines a collection \(A\) of instances of structured types or primitive values, based on the input set of the aggregate transformation, and a path \(p\) that occurs in the aggregate expression. Let \(p_1\) denote a data aggregation path~~An aggregate expression may be:~~

- ~~an expression valid in a `$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 ,~~
- ~~a e,~~
- ~~any of the above, followed by a expression,~~
- ~~any of the above, enclosed in parentheses and prefixed with a navigation path to related entities,~~
- ~~the virtual property .~~

~~Any aggregate expression that specifies~~ with single- or collection-valued segments and \(p_2\) a type-cast segment. Depending on its type, the aggregate expression contains a path \(p=p_1\) or \(p=p_2\) or \(p=p_1/p_2\). Each type of aggregate expression defines a function

\(f(A)\) which the aggregate transformation evaluates to obtain the property value.

The property is a dynamic property, except for a special case in type 4. In types 1 and 2, the aggregate expression MUST end with the keyword `with` and an aggregation method \(g\). The aggregation method also determines the type of the dynamic property. In types 1, 2, and 3 the aggregate expression MUST, and in type 4 it MAY, be followed by the keyword `as` and an alias, which is then the name of the dynamic property.

Types of aggregate expressions:

1. A path \(p=p_1\) or \(p=p_1/p_2\) where the last segment of \(p_1\) has a complex or entity or aggregatable primitive type~~MUST define an for the resulting~~ whose values can be aggregated using the specified aggregation method~~value. The resulting instance contains one dynamic property per parameter representing the aggregated value across all instances within~~ \(g\), or \(p=p_2\) if the input set can be aggregated using the custom aggregation method \(g\).
   Let \(f(A)=g(A)\).
2. An aggregatable expression whose values can be aggregated using the specified aggregation method \(g\).
   Let \(f(A)=g(B)\) where \(B\) is the collection consisting of the values of the aggregatable expression evaluated relative to each occurrence in \(A\) with null values removed from \(B\). In this type, \(p\) is absent.
3. A path \(p/{\tt\$count}\) (see section 3.2.1.4) with optional prefix \(p/{}\) where \(p=p_1\) or \(p=p_2\) or \(p=p_1/p_2\).
   Let \(f(A)\) be the cardinality of \(A\).
4. A path \(p/c\) consisting of an optional prefix \(p/{}\) with \(p=p_1\) or \(p=p_1/p_2\) where the last segment of \(p_1\) has a structured type or \(p=p_2\), and a custom aggregate~~. The JSON representation of these dynamic properties will include `odata.type` annotations where required by . If paths are present, the corresponding~~ \(c\) defined on the collection addressed by \(p\).
   Let \(f(A)=c(A)\). If computation of the custom aggregate fails, the service MUST reject the request. In the absence of an alias:
   - The name of the property is the name of the custom aggregate.
   - The property is a dynamic property whose type is determined by the custom aggregate, unless there is a declared property with that name. The latter case is allowed by the `CustomAggregate` annotation.

Determination of \(A\):

Let \(I\) be the input set. If \(p\) is absent, let \(A=I\) with null values removed.

Otherwise, let \(q\) be the portion of \(p\) up to and including the last navigation property, if any, and any type-cast segment that immediately follows, and let \(r\) be the remainder, if any, of \(p\) that contains no navigation properties, such that \(p\) equals the concatenated path \(q⁄r\). The aggregate transformation considers each entity reached via the path \(q\) exactly once. To this end, using the \(\Gamma\) notation:

- If \(q\) is non-empty, let \(E=\Gamma(I,q)\) and remove duplicates from that entity collection: If multiple representations of the same non-transient entity are reached, the service MUST merge them into one occurrence in \(E\) if they are complementary and MUST reject the request if they are contradictory. (See example 128.) If multiple occurrences of the same transient entity~~implicitly expanded to make the properties part of the result representation.~~ are reached, the service MUST keep only one occurrence in \(E\).
- If \(q\) is empty, let \(E=I\).

Then, if \(r\) is empty, let \(A=E\), otherwise let \(A=\Gamma(E,r)\), this consists of instances of structured types or primitive values, possibly with repetitions.

3.2.1.2 Keyword as

Aggregate expressions can be followed by the as keyword followed by an alias~~The~~ .

Example 7:

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

results in

```
{
  "@context": "$metadata#Sales(Total, MxA)",
  "value": [
    { "Total@type": "Decimal", "Total": 24,
      "MxA@type": "Decimal", "MxA": 8 }
  ]
}
```

Example 8:

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

results in

```
transformation{
  "@context": "$metadata#Sales(Tax)",
  "value": [
    { "Tax@type": "Decimal", "Tax": 2.08 }
  ]
}
```

An alias affects the structure of the ~~result set: An expression resulting in a simple value and a custom aggregate~~ output set: each alias corresponds to a dynamic property in a $select option. ~~If they are preceded by a navigation path, the corresponding `$select` option would be nested in one `$expand` option for each navigation property in the navigation path.~~

~~3.1.1 Keyword as~~

~~Aggregate expressions can define an alias using the `as` keyword, followed by a SimpleIdentifier (see **[, section 17.2]**).~~
~~The alias will introduce a dynamic property in the aggregated result set. The introduced dynamic property is added to the type containing the original expression or custom aggregate. The alias MUST NOT collide with names of declared properties, custom aggregates, or other aliases in that type.~~
~~When an is specified, an alias MUST be applied to the expression.~~

~~*Example :*~~

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

~~*results in*~~

```
{
```

```
  "@odata.context": "$metadata#Sales(Total,MxA)",
  "value": [
    { "@odata.id": null, "Total": 24, "MxA": 8 }
  ]
}
```

~~*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 :*~~

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

~~*results in*~~

3.2.1.3 Aggregation Methods

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

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

~~3.1.2 Keyword `with`~~

~~The keyword `with` is used to apply an to an or expression. The property or expression being aggregated is followed by the keyword `with`, followed by the name of the aggregation method to apply, followed by the keyword 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. Only types 1 and 2 of the aggregation algorithm~~Aggregate expressions containing an aggregation method MUST define an for the resulting aggregate value~~ involve aggregation methods, and the algorithm ensures that no null values occur among the values to be aggregated.

3.2.1.3.1 Standard Aggregation Method sum

~~3.1.3.1 Standard Aggregation Method `sum`~~

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

Example ~~:~~9:

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

results in

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

```
~~}~~
```

~~3.1.3.2 Standard Aggregation Method min~~

```
  ]
}
```

3.2.1.3.2 Standard Aggregation Method min

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

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

Example ~~:~~10:

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

results in

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

```
~~}~~
```

~~3.1.3.3 Standard Aggregation Method max~~

```
  ]
}
```

3.2.1.3.3 Standard Aggregation Method max

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

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

Example ~~:~~11:

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

results in

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

```
~~}~~
```

~~3.1.3.4 Standard Aggregation Method average~~

```
  ]
}
```

3.2.1.3.4 Standard Aggregation Method average

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

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

Example ~~:~~12:

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

results in

```
{
```

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

```
~~}~~
```

~~3.1.3.5 Standard Aggregation Method countdistinct~~

```
  ]
}
```

3.2.1.3.5 Standard Aggregation Method countdistinct

The aggregation method countdistinct ~~counts~~can be applied to arbitrary collections to count the distinct values~~, omitting any null values. For navigation properties, it counts~~. Instance comparison uses the ~~distinct entities~~definition of equality in ODataURL, section 5.1.1.1.1~~the union of all entities related to entities in the input set. For collection-valued primitive properties, it counts the distinct items in the union of all collection values in the input set. ~~.

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

Example ~~:~~13:

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

results in

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

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

3.2.1.3.6 Custom Aggregation Methods

~~3.1.3.6 Custom Aggregation Methods~~

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

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

⚠ *Example ~~:~~14: custom aggregation method~~s~~ that concatenates distinct string values separated by commas*

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

results in

```
{
  "@~~odata.~~context":
"$metadata#Sales(Customer(Country),~~Amount~~Total,ProductNames)"
,
  "value": [
~~  {"@odata.id":null,~~
      { "Customer~~":{~~": { "Country~~":"~~": "Netherlands" },
        "~~Amount":~~Total@type": "Decimal", "Total":  5,
        "ProductNames~~:"~~": "Paper,Sugar" },
~~  {"@odata.id":null,~~
      { "Customer~~":{~~": { "Country~~":"~~": "USA" },
        "~~Amount":~~Total@type": "Decimal", "Total": 19,
        "ProductNames~~:"~~": "Coffee,Paper,Sugar" }
~~]~~
~~}~~
  ]
}
```

3.2.1.4 Aggregate Expression $count

The aggregate expression $count is defined as type 3 in the aggregation algorithm. It MUST always specify an alias and MUST NOT specify an aggregation method.

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

Example 15:

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

results in

```
{
```

~~3.1.4 Keyword `from`~~

```
  "@context": "$metadata#Sales(SalesCount)",
  "value": [
    { "SalesCount@type": "Decimal", "SalesCount": 8 }
  ]
}
```

3.2.1.5 Keyword from

The from keyword ~~gives control over the order~~offers a shortcut for a sequence of groupby and aggregate~~aggregation across properties~~ transformations with the pattern \({\tt groupby}(…,{\tt aggregate}(…{\tt\ as\ }D_1))/{\tt aggregate}(D_1{\tt\ with\ }…)\).

In the following \(p_1,…,p_n\) are data aggregation paths that are allowed in groupby for simple grouping~~not part of the result structure and over the aggregation methods applied in every step.~~ .

1. If \(α\) is an aggregate expression and \(g\) is an aggregation method, then \[α{\tt\ from\ }p_1,…,p_n{\tt\ with\ }g\] is an aggregate expression which evaluates to the value of property \(D\) in the single instance in the output set of the following transformation sequence: \[{\tt groupby}((p_1,…,p_n),{\tt aggregate}(α{\tt\ as\ }D_1))/{\tt aggregate}(D_1{\tt\ with\ }g{\tt\ as\ }D).\]
2. If \(α=p/c{\tt\ from\ }…\) is an aggregate expression that starts with a custom aggregate \(c\), optionally prefixed with a path \(p\) as in type 4 in the aggregation algorithm, and that optionally continues with from and with clauses that were introduced through application of these rules, then \[α{\tt\ from\ }p_1,…,p_n\] is an aggregate expression which evaluates to the value of property \(c\) in the single instance in the output set of the following transformation sequence: \[{\tt groupby}((p_1,…,p_n),{\tt aggregate}(α{\tt\ as\ }D_1))/{\tt aggregate}(p/c).\]

Aggregate expressions constructed by these rules MUST be followed in the aggregate transformation by the keyword as and an alias. These rules can be applied repeatedly and lead to multiple from and with clauses in an aggregate expression.

⚠ Example 16~~Instead of applying a single aggregation method for calculating the aggregated value of an expression across all properties not included in the result structure, other aggregation methods to be~~

~~applied when certain properties MAY be specified using the `from` keyword, followed by a property path of a groupable property. Each groupable property MUST be followed by a `with` clause unless the aggregate expression is a custom aggregate, in which case the provider-defined behavior of the custom aggregate is used:~~

~~*aggregateExpression* as *alias*~~
~~from $groupableProperty_1$ [with $aggregationMethod_1$]~~
~~…~~
~~from $groupableProperty_n$ [with $aggregationMethod_n$]~~

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

~~If the `from` keyword is present, first the aggregation method determined by the aggregate expression is used to aggregate away properties that are not mentioned in a `from` clause and are not .~~

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

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

~~groupby(($groupableProperty_1$, …, $groupableProperty_n$),~~
~~aggregate(*aggregateExpression* as $tmpalias_1$))~~
~~/groupby(($groupableProperty_2$, …, $groupableProperty_n$),~~
~~aggregate($tmpalias_1$ with $aggregationMethod_1$ as $tmpalias_2$))~~
~~…~~
~~/groupby(($groupableProperty_n$),~~
~~aggregate($tmpalias_{n-1}$ with $aggregationMethod_{n-1}$ as $tmpalias_n$))~~
~~/aggregate($tmpalias_n$ with $aggregationMethod_n$ as *alias*))~~

~~The order of `from` clauses has to be compatible with hierarchies~~ referenced from a or specified as an ~~unnamed hierarchy in~~ : ~~l~~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 :*~~

: illustrates rule 1 where \(α={\tt Amount\ with\ sum}\), \(p_1={\tt Time}\), \(g={\tt average}\)

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

~~*is equivalent to*~~

```
                                             as DailyAverage)
```

is equivalent to (but avoids the intermediate dynamic property Total)

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

and results in the average sales volume per day

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

```
    { "DailyAverage@type": "Decimal", "DailyAverage":
3.428571428571429 }
~~]~~
```

```
~~}~~
```

~~3.1.5 Virtual Property $count~~

~~The value of the virtual property $count is the number of instances in the input set. It MUST always specify an and MUST NOT specify an .~~
~~The result property will have type Edm.Decimal with Scale="0" and sufficient Precision.~~

~~Example :~~

```
  ]
}
```

⚠ Example 17: illustrates rule 1 where \(α={\tt Forecast}\), \(p_1={\tt Time}\), \(g={\tt average}\)

```
GET ~~~/~~/service/Sales?$apply=aggregate~~($count as
SalesCount)~~(Forecast from Time with average
                                               as DailyAverage)
```

is equivalent to

~~results in~~

```
~~{~~
```

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

~~3.2 Transformation topcount~~

~~The topcount transformation takes two parameters.~~
~~The first parameter specifies the number of 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on instances of the input set and MUST result in a primitive numeric value.~~
~~The transformation retains the number of instances specified by the first parameter that have the highest values specified by the second expression.~~
~~In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.~~

~~Example :~~

```
GET
~~~/~~/service/Sales?$apply=~~topcount(2,~~groupby((Time),aggregate(Fore
cast))
```

```
                   /aggregate(Forecast with average as
DailyAverage)
```

⚠ Example 18: the maximal daily average for sales of a product

```
GET /service/Sales?$apply=aggregate(Amount with average from
Time,Product/Name
                                             with max as
MaxDailyAverage)
```

results in

```
{
  "@~~odata.~~context": "$metadata#Sales~~",~~(MaxDailyAverage)",
  "value": [
~~  { "ID": 3, "Amount": 4, ... },~~
~~  { "ID": 4, "Amount~~
    { "MaxDailyAverage@type": "Decimal", "MaxDailyAverage":
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 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on instances of the input set and MUST result in a primitive numeric value.~~
~~The transformation returns the minimum set of instances that have the highest values specified by the second parameter and whose sum of these values is equal to or greater than the value specified by the first parameter. It does not change the order of the instances in the input set.~~
~~In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.~~

~~Example :~~

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

~~results in~~

```
~~{~~
```

```
~~  "@odata.context": "$metadata#Sales",~~
~~  "value": [~~
~~    { "ID": 3, "Amount": 4, ... },~~
~~    { "ID": 4, "Amount": 8, ... },~~
~~    { "ID": 5, "Amount": 4, ... }~~
~~  ]~~
~~}~~
```

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

~~3.4 Transformation toppercent~~

~~The toppercent transformation takes two parameters.~~
~~The first parameter specifies the number of 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on instances of the input set and MUST result in a primitive numeric value.~~
~~The transformation returns the minimum set of instances that have the highest values specified by the second parameter and whose cumulative total is equal to or greater than the percentage of the cumulative total of all instances in the input set specified by the first parameter. It does not change the order of the instances in the input set.~~
~~In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.~~

~~Example :~~

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

~~results in~~

```
~~{~~
```

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

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

~~3.5 Transformation bottomcount~~

~~The bottomcount transformation takes two parameters.~~
~~The first parameter specifies the number of 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 instances are compared for determining the result set. It MUST be an expression that can be evaluated on instances of the input set and MUST result in a primitive numeric value.~~
~~The transformation retains the number of instances specified by the first parameter that have the lowest values specified by the second parameter. It does not change the order of the instances in the input set.~~
~~In case the value of the second expression is ambiguous, the service MUST impose a stable ordering before determining the returned instances.~~

~~Example :~~

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

~~results in~~

```
~~{~~
```

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

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

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

~~3.6 Transformation `bottomsum`~~

~~The `bottomsum` transformation takes two parameters.~~

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

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

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

~~Example :~~

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

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

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

~~Example :~~

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

~~*results in*~~

```
~~{~~
```

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

```
    ]
}
```

3.2.2 Transformation concat

```
~~  "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 :*~~

```
~~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 transformation sequence to the input set and concatenates the intermediate ~~result~~output sets in the order of the parameters into the ~~result~~output set, preserving the ordering of the individual ~~result~~output sets as well as the structure of each ~~result~~ instance in these sets, potentially leading to ~~an inhomogeneously~~a non-homogeneously structured ~~result~~output set. If different intermediate output sets contain dynamic properties with the same alias, clients SHOULD ensure they have the same type and meaning in each intermediate output set.

⚠ *Example ~~:~~19:*

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

results in

```
{
```

~~*results in*~~

```
~~{~~
```

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

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

3.2.3 Transformation groupby

~~The result set of `concat` has a 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 initial set into subsets~~ where ~~all instances in a subset have the same values for the grouping properties specified in the first parameter,~~

~~2. Applies~~ the second is a list of set transformations, separated by forward slashes to ~~each subset according to the second parameter, resulting in a new set of potentially different structure and cardinality,~~

~~Ensures~~express that they are consecutively applied. If the second parameter is not specified, it defaults to a single transformation whose output set consists of a single instance of the input type~~the instances in the result set contain all grouping properties with the correct values for the group,~~ without properties and without entity id.

3.2.3.1 Simple Grouping

~~3. 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 parenthesized list \(G\) of one or more data aggregation paths with single-valued ~~property paths (paths ending in a single-valued primitive, complex, or navigation property) that is enclosed in parentheses. The same property~~segments. The same path SHOULD NOT appear more than once; redundant property paths MAY be considered valid, but MUST NOT alter the meaning of the request. Navigation properties and stream properties specified in grouping properties are expanded by default (see example 72~~If the property path leads to a single-valued navigation property, this means grouping by the entity-id of the related entities.~~).

The algorithmic description of this transformation makes use of the following definitions: Let \(u[q]\) denote the value of a structural or navigation property \(q\) in an instance \(u\). A path \(p_1\) is called a *prefix* of a path \(p\) if there is a non-empty path \(p_2\) such that \(p\) equals the concatenated path \(p_1/p_2\). Let \(e\) denote the empty path.

The output set of the groupby transformation is constructed in five steps.

1. For each occurrence \(u\) in the input set, a projection is computed that contains only the grouping properties. This projection is \(s_G(u,e)\) and the function \(s_G(u,p)\) takes an instance and a path relative to the input set as arguments and is computed recursively as follows:
    - Let \(v\) be an instance of the type of \(u\) without properties and without entity id.
    - For each structural or navigation property \(q\) of \(u\):
        - If \(u\) has a subtype of the type addressed by \(p\) and \(q\) is only declared on that subtype, let \(p'=p/p''/q\) where \(p''\) is a type-cast to the subtype, otherwise let \(p'=p/q\).
        - If \(p'\) occurs in \(G\), let \(v[q]=u[q]\).
        - Otherwise, if \(p'\) is a prefix of a path in \(G\), let \(v[q]=s_G(u[q],p')\).
    - Return \(v\).
2. The input set is split into subsets where two instances are in the same subset if their projections are the same. If representations of the same non-transient entity are encountered during the comparison of two projections, the service MUST assign them to one subset with the merged representation if they are complementary and MUST reject the request if they are contradictory.
3. The set transformations from the second parameter are applied to each subset, resulting in a new set of potentially different structure and cardinality. Associated with each resulting set is the common projection of the instances in the subset from which the resulting set was computed.

4. Each set resulting from the previous step is transformed to contain the associated common projection \(s\). This transformation is denoted by \(\Pi_G(s)\) and is defined below.
5. The output set is the concatenation of the transformed sets from the previous step. The order of occurrences from the same transformed set remains the same, and no order is defined between occurrences from different transformed sets.

Definition of \(\Pi_G(s)\):

Prerequisites: \(G\) is a list of data aggregation paths and \(s\) is an instance of the input type.

The output set of the transformation \(\Pi_G(s)\) is in one-to-one correspondence with its input set via the order-preserving~~The optional second parameter is a list of set transformations, separated by forward slashes to express that they are consecutively applied. Transformations may take into account the grouping properties for producing their result, e.g. `aggregate` removes properties that are used neither for grouping nor for aggregation.~~

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

mapping \(u↦a_G(u,s,e)\). The function \(a_G(u,s,p)\) takes two instances and a path relative to the input set as arguments and is computed recursively as follows:

1. If necessary, cast \(u\) to a subtype so that its type contains all structural and navigation properties of \(s\).
2. For each structural or navigation property \(q\) of \(s\):
   - If \(s\) has a subtype of the type addressed by \(p\) and \(q\) is only declared on that subtype, let \(p'=p/p''/q\) where \(p''\) is a type-cast to the subtype, otherwise let \(p'=p/q\).
   - If \(q\) is a single-valued primitive structural property or \(p'\) occurs in \(G\), let \(u[q]=s[q]\). (In the case where \(p'\) occurs in \(G\) we also call \(q\) a *final segment from \(G\)*.)
   - Otherwise, if \(q\) is single-valued, let \(u[q]=a_G(u[q],s[q],p')\).
   - Otherwise, the behavior is undefined. (Such cases never occur when \(\Pi_G(s)\) is used in this document.)
3. Return \(u\).

Example ~~:~~20:

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

~~results in~~

```
~~{~~
```

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

results in

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

```
~~  ]~~
~~}~~
```

```
~~The~~       "Total@type": "Decimal", "Total":  2 }
  ]
}
```

If the second parameter ~~can be~~is omitted~~ to request distinct value combinations of~~, steps 2 and 3 above produce one instance containing only the grouping properties~~.~~ per distinct value combination.

Example ~~:~~21:

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

results in

```
{
```

~~and result in~~

```
~~{~~
```

```
~~"@odata.~~ "@context": "$metadata#Sales(Product(Name),Amount)",
  "value": [
~~{"@odata.id": null,~~
    { "Product": { "Name": "Coffee" }, "Amount": 4 },
~~{"@odata.id": null,~~
    { "Product": { "Name": "Coffee" }, "Amount": 8 },
~~{"@odata.id": null,~~
    { "Product": { "Name": "Paper"  }, "Amount": 1 },
~~{"@odata.id": null,~~
    { "Product": { "Name": "Paper"  }, "Amount": 2 },
~~{"@odata.id": null,~~
    { "Product": { "Name": "Paper"  }, "Amount": 4 },
~~{"@odata.id": null,~~
    { "Product": { "Name": "Sugar"  }, "Amount": 2 }
~~]~~
  ]
}
```

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

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

A groupby transformation affects the structure of the ~~result~~output set similar to $select where each grouping property corresponds to an item in a $select clause. ~~Grouping properties that specify navigation properties are automatically expanded, and the specified properties of that navigation property correspond to properties specified in a $select expand option on the expanded navigation property. The set transformations specified in the second parameter of groupby further affect the structure as described for each transformation; for example, the transformation adds properties for each aggregate expression.~~

3.2.3.2 Grouping with rollup

~~3.10.2 Grouping with rollup and $all~~

The rollup grouping operator allows applying set transformations to instances of an input set organized in a leveled hierarchy~~requesting additional levels of aggregation in addition to the most granular level defined by the grouping properties.~~. It can be used instead of a grouping property ~~path~~ in the first parameter of groupby.

~~The rollup grouping operator~~ It 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 prefixed with the navigation path leading to the annotated entity type. This named hierarchy is used for grouping instances.~~

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

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

~~Grouping~~A groupby with rollup applied to a leveled hierarchy allows requesting aggregation for all levels of that hierarchy. It splits the input set into groups using all grouping properties (see (1) below), then removes the last property from the hierarchy (see (2)) and repeats this process using the remaining grouping properties until all of the levels have been used up (see terminating rule (3)).

Such a grouping with rollup for a leveled hierarchy is processed ~~for leveled hierarchies ~~using the following equivalence relationships, in which ~~p_i is a~~ \(p_1,…,p_k\) are groupable property ~~path, T~~paths representing a level, \(T\) is a transformation sequence, the ellipsis \(…\) stands in for zero or more property paths, ~~and R~~\(P_1\) stands in for zero or more property paths and \(P_2\) for zero or more rollup or rolluprecursive operators or property paths:

- \({\tt groupby~~((~~}((P_1,{\tt rollup~~$(p_1, \ldots, p_{n-1}, p_n), R), T)$~~}(p_1,…,p_{k-1},p_k),P_2),T)\) is equivalent to \[\matrix{ {\tt concat~~(~~}(\hfill\\ \quad {\tt groupby~~$((p_1, \ldots, p_{n-1}, p_n, R), T)$,~~}((P_1,p_1,…,p_{k-1},p_k,P_2),T),\hfill&\tt (1)\\ \quad {\tt groupby~~((~~}((P_1,{\tt rollup~~$(p_1, \ldots, p_{n-1}), R), T))$~~}(p_1,…,p_{k-1}),P_2),T)\hfill&\tt(2)\\ ).\hskip25pc\\ }\]
- \({\tt groupby~~((~~}((P_1,{\tt rollup~~$(p_1, p_2), R), T)$~~}(p_1,p_2),P_2),T)\) is equivalent to \[\matrix{ {\tt concat~~(groupby($(p_1, p_2, R), T)$,~~}(\hfill&\tt (3)\\ \quad {\tt groupby~~$((p_1, R), T))$~~}((P_1,p_1,p_2,P_2),T),\hfill\\ \quad {\tt groupby}((P_1,p_1,P_2),T)\hfill\\ ).\hskip25pc\\ }\]
- ~~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 ~~:~~22: rolling up two hierarchies, the first with two levels, the second with three levels:

~~(~~ \[({\tt rollup~~$(p_{1,1}, p_{1,2})$,~~}(p_{1,1},p_{1,2}),{\tt rollup~~$(p_{2,1}, p_{2,2}, p_{2,3}))$~~

}(p_{2,1},p_{2,2},p_{2,3}))\] will result in the six groupings \[\matrix{
(p_{1,1},p_{1,2},\hfill&p_{2,1},p_{2,2},p_{2,3})\hfill\\
(p_{1,1},p_{1,2},\hfill&p_{2,1},p_{2,2})\hfill\\ (p_{1,1},p_{1,2},\hfill&p_{2,1})\hfill\\
(p_{1,1},\hfill&p_{2,1},p_{2,2},p_{2,3})\hfill\\ (p_{1,1},\hfill&p_{2,1},p_{2,2})\hfill\\
(p_{1,1},\hfill&p_{2,1})\hfill }\] The leveled hierarchy of the first rollup has 2 levels, the one of the second has 3 levels, and the groupings represent all possible \(6=2·3\) combinations of levels from both hierarchies.

~~$(p_{1,1}, p_{1,2},$ $p_{2,1}, p_{2,2}, p_{2,3})$~~
~~$(p_{1,1}, p_{1,2},$ $p_{2,1}, p_{2,2})$~~
~~$(p_{1,1}, p_{1,2},$ $p_{2,1})$~~
~~$(p_{1,1},$ $p_{2,1}, p_{2,2}, p_{2,3})$~~
~~$(p_{1,1},$ $p_{2,1}, p_{2,2})$~~
~~$(p_{1,1},$ $p_{2,1})$~~

~~Note that `rollup` stops one level earlier than `GROUP BY ROLLUP` in TSQL, see , unless the virtual property `$all` is used as the hierarchy root level. Loosely speaking the root level is never rolled up. Ordering of rollup instances within detail instances is up to the service if no `$orderby` is given, otherwise at the position determined by `$orderby`.~~

Example ~~:~~23: answering the second question in section 2.3~~section~~

```
GET
~~~/~~/service/Sales?$apply=groupby((rollup(Customer/Country,Cust
omer/Name),

rollup(Product/Category/Name,Product/Name~~),~~
~~                 Currency/Code),~~
~~                 aggregate(Amount with sum as Total))~~)),
                                             aggregate(Amount with sum
as Total))
```

results in seven entities for the finest grouping level

```
{
~~"@odata.~~  "@context~~":"$~~":
"$metadata#Sales(Customer(Country~~,Name),~~),

Product(Category(Name~~),Name),~~)),Total~~,Currency(Code))",~~)",
  "value": [
~~  { "@odata.id": null,~~
    { "Customer": { "Country": "USA", "Name": "Joe" },
      "Product":  { "Category": { "Name": "Non-Food" },
"Name": "Paper" },
      "Total@type": "Decimal", "Total": 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" },
      "Total~~":~~@type": "Decimal", "Total":  2~~, "Currency": { "Code":
"USD" }~~
~~  },~~
~~  ...~~ },
    ...
```

six without product name

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

and four with neither customer nor product name

```
    {~~ "@odata.id": null,~~ "Customer": { "Country": "USA" },
      "Product":  { "Category": { "Name": "Food" } },
      "Total@type": "Decimal", "Total": 14~~, "Currency": { "Code":
"USD" }~~
~~  },~~
~~  ...~~
~~]~~ },
    ...
  ]
}
```

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

If rollup is used with one parameter, the parameter references a named leveled hierarchy to be used for grouping instances, and therefore MUST be the value of the Qualifier attribute of an annotation with term LeveledHierarchy. If the annotation has qualifier \(Q\) and as value a collection consisting of \(p_1,…,p_n\) with \(n≥2\), then \({\tt rollup}(Q)\) is equivalent to \({\tt rollup}(p_1,…,p_n)\).

Another grouping operator rolluprecursive which similarly works with a recursive hierarchy is defined later.

3.3 Transformations Producing a Subset

These transformations produce an output set that is a subset of their input set, possibly in a different order. Some of the algorithmic descriptions below make use of the following definition: A total order of a collection is called *stable across requests* if it is the same for all requests that construct the collection by executing the same resource path and transformations, possibly nested, on the same underlying data.

⚠ Example 24: A stable total order is required for the input set of a `skip` transformation. The following request constructs that input set by executing the `groupby` transformation on the `Sales` entity collection, computing the total sales per customer. Because of the subsequent `skip` transformation, the service must endow this with a stable total order. Then the request divides the total sales per customer into pages of \(N\) customers and returns page number \(i\) in a reproducible manner (as long as the underlying data do not change).

```
GET /service/Sales?$apply=
  groupby((Customer),aggregate(Amount with sum as Total))
  /skip(M)/top(N)
```

where the number in `skip` is \(M=(i-1)·N\). Other values of \(M\) can be used to skip, for example, half a page.

3.3.1 Top/bottom transformations

These transformations take two parameters. The first parameter MUST be an expression that is evaluable on the input set as a collection, without reference to an individual instance (and which therefore cannot be a property path). The second parameter MUST be an expression that is evaluated on each instance of the input set in turn.

The output set is constructed as follows:

1. Let \(A\) be a copy of the input set with a total order that is chosen by the service (it need not preserve any existing order). The total order MUST be stable across requests. (This is the order of the eventual output set of this transformation.)
2. Let \(B\) be a copy of \(A\) that is stable-sorted in ascending (for transformations starting with `bottom`) or descending (for transformations starting with `top`) order of the value specified in the second parameter. (This is the order in which contributions to the output set are considered.)
3. Start with an empty output set.
4. Loop over \(B\) in its total order.
5. Exit the loop if a condition is met. This condition depends on the transformation being executed and is given in the subsections below.
6. Insert the current item of the loop into the output set in the order of \(A\).
7. Continue the loop.

For example, if the input set consists of non-transient entities and the datastore contains an index ordered by the second parameter and then the entity id, a service may implement this algorithm with \(A=B\) ordered like this index.

The order of the output set can be influenced with a subsequent orderby transformation.

3.3.1.1 Transformations bottomcount and topcount

The first parameter MUST evaluate to a positive integer \(c\). The second parameter MUST evaluate to a primitive type whose values are totally ordered. In step 5, exit the loop if the cardinality of the output set equals \(c\).

Example 25:

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

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": 1, "Amount": 1 },
    { "ID": 7, "Amount": 1 }
  ]
}
```

Example 26:

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

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": 3, "Amount": 4 },
    { "ID": 4, "Amount": 8 }
  ]
}
```

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 in step 1. Such a logic needs to be in place even with a preceding orderby since it cannot be ensured that it creates a stable order of the instances on the expressions of the second parameter.

3.3.1.2 Transformations bottompercent and toppercent

The first parameter MUST evaluate to a positive number \(p\) less than or equal to 100. The second parameter MUST evaluate to a number. In step 5, exit the loop if the ratio of the sum of the numbers addressed by the second parameter in the output set to their sum in the input set equals or exceeds \(p\) percent.

Example 27:

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

results in

```
{
  "@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 }
  ]
}
```

Example 28:

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

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": 3, "Amount": 4 },
    { "ID": 4, "Amount": 8 }
  ]
}
```

3.3.1.3 Transformations bottomsum and topsum

The first parameter MUST evaluate to a number \(s\). The second parameter MUST be an aggregatable expression that evaluates to a number. In step 5, exit the loop if the sum of the numbers addressed by the second parameter in the output set is greater than or equal to \(s\).

Example 29:

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

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": 1, "Amount": 1 },
    { "ID": 2, "Amount": 2 },
    { "ID": 6, "Amount": 2 },
    { "ID": 7, "Amount": 1 },
    { "ID": 8, "Amount": 2 }
  ]
}
```

Example 30:

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

results in

```
{
  "@context": "$metadata#Sales",
```

~~3.11~~ "value": [~~Transformation filter~~

```
    { "ID": 3, "Amount": 4 },
    { "ID": 4, "Amount": 8 },
    { "ID": 5, "Amount": 4 }
  ]
}
```

3.3.2 Transformation filter

The filter transformation takes a Boolean expression that could also be passed as a $filter system query option~~ to its~~. Its output set is the subset of the input set ~~and returns~~containing all instances (possibly with repetitions) for which this expression~~ evaluates to~~, evaluated relative to the instance, yields true. No order is defined on the output set.

Example ~~:~~31:

GET ~~~/~~/service/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 that could also be passed as a `$expand` system query option as its first parameter. The optional second parameter can be either a transformation that will be applied to the 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 according to the specified expand options.~~

~~*Example :*~~

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

~~*results in*~~

```
~~{~~
```

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

~~*The result has the same structure and content as*~~

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

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

~~*Example : nested `expand` transformations*~~

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

results in

```
{
  "@~~odata.~~context": "$metadata#~~Customers~~Sales",
```

```
~~  "value": [~~
```

```
  "value": [
```

```
    { "ID": ~~"PC1", "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~~, … },~~ }
  ]
}
```

3.3.3 Transformation orderby

The orderby transformation takes a list of expressions that could also be passed as a $orderby system query option. Its output set consists of the instances of the input set in the same order $orderby would produce for the given expressions, but keeping the relative order from the input set if the given expressions do not distinguish between two instances. The orderby transformation thereby performs a stable-sort. A service supporting this transformation MUST at least offer sorting by values addressed by property paths, including dynamic properties, with both suffixes asc and desc.

Example 32:

```
GET /service/Sales?$apply=groupby((Product/Name),
                            aggregate(Amount with sum as
Total))
                   /orderby(Total desc)
```

```
results in~~                    { "ID": 8, "Amount": 2, ... } ] },~~
~~          { "ID": "P4", "Name": "Pencil", "Color": "Black", "Sales": [] }~~
~~        ]~~
~~      }~~
~~    ]~~
~~}~~
```

~~3.13 Transformation search~~

```
{
  "@context": "$metadata#Sales(Product(Name),Total)",
  "value": [
    { "Product": { "Name": "Coffee" },
      "Total@type": "Decimal", "Total": 12 },
    { "Product": { "Name": "Paper" },
      "Total@type": "Decimal", "Total":  8 },
    { "Product": { "Name": "Sugar" },
      "Total@type": "Decimal", "Total":  4 }
  ]
}
```

3.3.4 Transformation search

The search transformation takes a search expression that could also be passed as a $search system query option~~ to its~~. Its output set is the subset of the input set ~~and returns~~containing all ~~entities~~instances (possibly with repetitions) that match this search expression. Closing parentheses in search expressions must be within single or double quotes in order to avoid syntax errors like search()). No order is defined on the output set.

Example ~~:~~33: assuming that free-text search on Sales takes the related product name into account,

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

results in

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

3.3.5 Transformation skip

The skip transformation takes a non-negative integer \(c\) as argument. Let \(A\) be a copy of the input set with a total order that extends any existing order of the input set but is otherwise chosen by the service. The total order MUST be stable across requests.

The transformation excludes from the output set the first \(c\) occurrences in \(A\). It keeps all remaining instances in the same order as they occur in \(A\).

Example 34:

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

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": 6, "Amount": 2 },
    { "ID": 7, "Amount": 1 }
  ]
}
```

3.3.6 Transformation top

The top transformation takes a non-negative integer \(c\) as argument. Let \(A\) be a copy of the input set with a total order that extends any existing order of the input set but is otherwise chosen by the service. The total order MUST be stable across requests.

If \(A\) contains more than \(c\) instances, the output set consists of the first \(c\) occurrences in \(A\). Otherwise, the output set equals \(A\). The instances in the output set are in the same order as they occur in \(A\).

Note the transformation top(0) produces an empty output set.

Example 35:

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

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": 4, "Amount": 8 },
    { "ID": 5, "Amount": 4 }
  ]
}
```

3.3.7 Stable Total Order Before $skip and $top

When the system query options $top and $skip OData-Protocol, sections 11.2.6.3 and 11.2.6.4 are executed after the system query option $apply and after $filter and $orderby, if applicable, they operate on a collection with a total order that extends any existing order but is otherwise chosen by the service. The total order MUST be stable across requests.

3.4 One-to-One Transformations

These transformations produce an output set in one-to-one correspondence with their input set. The output set is initially a clone of the input set, then dynamic properties are added to the output set. The values of properties copied from the input set are not changed, nor is the order of instances changed.

3.4.1 Transformation identity

```
~~}~~
```

The ~~result~~output set of ~~search~~ ~~has~~ the ~~same structure as the~~ identity transformation is its input set in unchanged order.

Example 36: Add a grand total row to the Sales result set

```
GET /service/Sales?$apply=concat(identity,aggregate(Amount
with sum as Total))
```

3.4.2 Transformation compute

~~3.14~~ The compute transformation takes a comma-separated list of one or more *compute expressions* as parameters.~~Transformation compute~~

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

A compute expression is ~~an~~a common expression ~~valid in a $filter system query option on the input set that *results in* a simple value,~~ followed by the as keyword, followed by an alias~~a SimpleIdentifier (see **[, section 17.2]**), called an alias. This alias MUST NOT collide with names of properties in the input set or with other aliases introduced in the same compute transformation.~~.

The ~~result~~output set is constructed by copying the instances of the input set and adding one dynamic property per compute expression to each occurrence~~each instance of~~ in the ~~in~~output set. The name of ~~the~~each added dynamic property is the alias ~~following the as keyword. The value of the property is the value of the~~of the corresponding compute expression~~evaluated on~~. The value of each added dynamic property is computed relative to the corresponding instance. Services MAY support expressions that ~~instance.~~address dynamic properties added by other expressions within the same compute transformation, provided that the service can determine an evaluation sequence. The type of the property is determined by the rules for evaluating ~~$filter~~common expressions and numeric promotion defined in OData-URL, section 5.1.1~~**[OData-URL]**. The JSON representation of these dynamic properties will include odata.type annotations where required by .~~.

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

Example ~~:~~37:

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

results in

```
{
```

~~results in~~

```
{
```

```
~~"@odata.~~  "@context": "$metadata#Sales~~",~~(*,Tax)",
  "value": [
    { "ID": 1, ~~…,~~"Amount": 1, "Tax@type": "Decimal", "Tax":
0.14 },
    { "ID": 2, ~~…,~~"Amount": 2, "Tax@type": "Decimal", "Tax":
0.12 },
    { "ID": 3, ~~…,~~"Amount": 4, "Tax@type": "Decimal", "Tax":
0.24 },
    { "ID": 4, ~~…,~~"Amount": 8, "Tax@type": "Decimal", "Tax":
0.48 },
    { "ID": 5, ~~…,~~"Amount": 4, "Tax@type": "Decimal", "Tax":
0.56 },
    { "ID": 6, ~~…,~~"Amount": 2, "Tax@type": "Decimal", "Tax":
0.12 },
    { "ID": 7, ~~…,~~"Amount": 1, "Tax@type": "Decimal", "Tax":
0.14 },
    { "ID": 8, ~~…,~~"Amount": 2, "Tax@type": "Decimal", "Tax":
0.28 }
  ]
}
```

3.4.3 Transformation addnested

The addnested transformation expands a path relative to the input set, applies one or more transformation sequences to the addressed resources, and adds the transformed resources as dynamic (navigation) properties to the output set. The output set \(A\) is initially a clone of the input set.

The first parameter of the addnested transformation is a path \(p\) or a concatenated path \(p/q\). Here, \(p=p_1/…/p_k\) with \(k≥1\) is a data aggregation path with single- or collection-valued segments. The path \(p\) MUST NOT contain any navigation properties prior to the last segment \(p_k\), which MUST either be a navigation or a complex structural property. If the optional \(q\) is present, it MUST be a type-cast segment. This is an extension of the definition in OData-URL, section 5.1.3 in that the first parameter need not contain a navigation property.

Further parameters are one or more transformation sequences followed by the as keyword followed by an alias whose name need not differ from names in the input set but MUST differ from names already in \(\Gamma(A,p_1/…/p_{k-1})\) (using the \(\Gamma\) notation) as well as from aliases for other transformation sequences.

If \(p_k\) is single-valued, the transformation sequences MUST consist of only identity or compute or addnested transformations, because these transform one-element collections into one-element collections. This makes it meaningful to speak (in this section only) of a transformation sequence applied to a single instance; this means applying it to a collection containing the single instance and taking as result the single instance from the output set.

For each occurrence \(u\) in \(\Gamma(A,p_1/…/p_{k-1})\), let \(B=γ(u,p_k/q)\) and let the resource \(v\) be

- the collection \(B\) if \(p_k\) is collection-valued
- the single instance in \(B\) if \(p_k\) is single-valued and \(B\) is non-empty
- undefined if \(p_k\) is single-valued and \(B\) is empty.

If \(v\) is defined, then for each transformation sequence, a dynamic property is added to \(u\) as follows: If \(p_k\) is a navigation property, the added property is a dynamic navigation property, which is expanded by default, otherwise it is a dynamic structural property. Its name is the alias of the transformation sequence. The value of the added property is the result of the transformation sequence applied to \(v\). The dynamic property carries as control information the context URL of \(v\).

Example 38:

```
GET /service/Customers?$apply=addnested(Sales,
                                        filter(Amount gt 3)
as FilteredSales)
```

results in

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

If Sales was a collection-valued complex property of type SalesModel.SalesComplexType, the context would be "FilteredSales@context": "#Collection(SalesModel.SalesComplexType)".

3.5 Transformations Changing the Input Set Structure

The output set of the join transformations differs from their input set in the number of instances as well as in their structure, but reflects the order of the input set. Transformation nest produces a one-instance output set.

3.5.1 Transformations join and outerjoin

The join and outerjoin transformations take as their first parameter \(p\) a collection-valued complex or navigation property, optionally followed by a type-cast segment to address only instances of that derived type or one of its sub-types, followed by the as keyword, followed by an alias. The optional second parameter specifies a transformation sequence \(T\).

For each occurrence \(u\) in an order-preserving loop over the input set

1. the instance collection \(A\) addressed by \(p\) is identified.
2. If \(T\) is provided, \(A\) is replaced with the result of applying \(T\) to \(A\).
3. In case of an outerjoin, if \(A\) is empty, a null instance is added to it.
4. For each occurrence \(v\) in an order-preserving loop over \(A\) an instance \(w\) is appended to the output set of the transformation:
    - The instance \(w\) is a clone of \(u\) with an additional dynamic property whose name is the given alias and whose value is \(v\).
    - The dynamic property is a navigation property if \(p\) is a collection-valued navigation property, otherwise it is a complex property.
    - The dynamic property carries as control information the context URL of \(v\).

Example 39: all links between products and sales instances

```
GET /service/Products?$apply=join(Sales as
Sale)&$select=ID&$expand=Sale
```

results in

```
{
  "@context": "$metadata#Products(ID,Sale())",
  "value": [
    { "ID": "P1",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 2, "Amount": 2 } },
    { "ID": "P1",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 6, "Amount": 2 } },
    { "ID": "P2",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 3, "Amount": 4 } },
    { "ID": "P2",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 4, "Amount": 8 } },
    { "ID": "P3",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 1, "Amount": 1 } },
    { "ID": "P3",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 5, "Amount": 4 } },
    { "ID": "P3",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 7, "Amount": 1 } },
    { "ID": "P3",
      "Sale": {
        "@context": "#Sales/$entity",
        "ID": 8, "Amount": 2 } }
  ]
}
```

In this example, $expand=Sale is used to include the target entities in the result. There are no subsequent transformations like groupby that would cause it to be expanded by default. If the first parameter Sales was a collection-valued complex property of type SalesModel.SalesComplexType, the complex property Sale would be in the result regardless, and its context would be "@context": "#SalesModel.SalesComplexType".

Applying outerjoin instead would return an additional instance for product with "ID": "P4" and Sale having a null value.

3.5.2 Transformation nest

The nest transformation takes as parameters one or more transformation sequences followed by the as keyword followed by an alias.

The output set consists of a single instance of the input type without entity id having one dynamic property per transformation sequence. The name of the dynamic property is the alias for this transformation sequence. The value of the dynamic property is the collection resulting from the transformation sequence applied to the input set. The dynamic property carries as control information the context URL of the transformed input set.

Example 40:

```
GET /service/Sales?$apply=nest(groupby((Customer/ID)) as
Customers))
```

results in

```
{
  "@context":"$metadata#Sales(Customers())",
  "value": [
    { "Customers@context": "#Sales(Customer(ID))",
      "Customers": [ { "Customer": { "ID": "C1" } },
                     { "Customer": { "ID": "C2" } },
                     { "Customer": { "ID": "C3" } } ] }
  ]
}
```

3.6 Expressions Evaluable on a Collection

The following two subsections introduce two new types of expression that are evaluated relative to a collection, called the input collection.

These expressions are

- either prepended with a collection-valued path \(p\) followed by a forward slash, like a lambda operator OData-URL, section 5.1.1.13. The collection identified by that path is then the input collection for the expression.
- or prepended with the keyword $these followed by a forward slash, the input collection is then the *current collection* defined as follows:
  - In a system query option other than $apply, possibly nested within $expand or $select, the current collection is the collection that is the subject of the system query option.
  - In a path segment that addresses a subset of a collection OData-URL, section 4.12, the current collection is the collection that is the subject of the path segment.
  - In an $apply transformation, the current collection is the input set of the transformation.

3.6.1 Function aggregate

The aggregate function allows the use of aggregated values in expressions. It takes a single parameter accepting an aggregate expression and returns the aggregated value of type Edm.PrimitiveType as the result from applying the aggregate expression on its input collection.

More precisely, if \(α\) is an aggregate expression, the function \(p/{\tt aggregate}(α)\) or \({\tt\$these}/{\tt aggregate}(α)\) evaluates to the value of the property \(D\) in the single instance of the output set that is produced when the transformation \({\tt aggregate}(α{\tt\ as\ }D)\) is applied with the input collection as input set.

Example 41: Sales making up at least a third of the total sales amount.

```
GET /service/Sales?$filter=Amount mul 3 ge
$these/aggregate(Amount with sum)
```

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": "4", "Amount": 8 }
  ]
}
```

Example 42: Products with more than 1.00 sales tax. The aggregate expression of type 2 combines paths with and without $it prefix (compare this with example 8).

```
GET /service/Products?$filter=Sales/aggregate(Amount mul
$it/TaxRate with sum)
                              gt 1
```

⚠ Example 43: products with a single sale of at least twice the average sales amount

```
GET /service/Products?$filter=Sales/any(s:s/Amount ge
                              Sales/aggregate(Amount with
average) mul 2)
```

Both examples result in

```
{
  "@context": "$metadata#Products",
  "value": [
    { "ID": "P3", "Name": "Paper", "Color": "White",
"TaxRate": 0.14 }
  ]
}
```

3.6.2 Expression $count

The expression $count evaluates to the cardinality of the input collection.

Example 44: The input collection for $count consists of all sales entities, the top third of sales entities by amount form the result.

```
GET /service/Sales?$apply=topcount($these/$count div
3,Amount)
```

results in 2 (a third of 8, rounded down) entities. (This differs from toppercent(33.3,Amount), which returns only the sales entity with ID 4, because that already makes up a third of the total amount.)

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": 3, "Amount": 4 },
    { "ID": 4, "Amount": 8 }
  ]
}
```

~~3.15~~ A definition that is equivalent to a $count expression after a collection-valued path was made in OData-URL, section 4.8~~Filter Function isdefined~~

.

3.7 Function isdefined

Properties that are not explicitly mentioned in aggregate or groupby are considered to have been *aggregated away*~~ and~~. Since they are treated as having the null value in $filter expressions OData-URL, section 5.1.1.15~~.~~, the $filter expression Product eq null cannot distinguish between an instance containing the value for the null product and the instance containing the aggregated value across all products (where the Product has been aggregated away).

The ~~filter~~ function isdefined can be used to determine whether a property ~~has been aggregated away.~~is present or absent in an instance. It takes a single-valued property path~~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~~is present in the instance for which the expression containing the isdefined function call is evaluated. A present property can still have the null value; it can represent a grouping of null values, or an aggregation that results in a null value.

Example ~~:~~45: Product has been aggregated away, causing an empty result

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

results in

```
{
  "@~~odata.~~context": "$metadata#Sales(Total)",
  "value": []
}
```

3.8 Evaluating $apply as an Expand and Select Option

~~3.16 Evaluating $apply~~

The new system query option ~~is evaluated first, then the other system query options are evaluated, if applicable, on the result of $apply in their normal order (see **[, section 11.2.1]**). If the result is a collection, $filter, $orderby, $expand and $select work as usual on properties that are defined on the output set after evaluating .~~
~~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 $orderby as having a value that is even lower than null, i.e. instances for which a property has been rolled up appear before instances that have a null value for that property when ordering ascending.~~
~~On resource paths ending in /$count the system query option is evaluated on the set identified by the resource path without the /$count segment, the result is the plain-text number of items in the result of . This is similar to the combination of /$count and $filter.~~
~~The $count system query option is evaluated after , the annotation @odata.count contains the number of items in the result of .~~
~~Providers MAY support $count, $top and $skip together with , in which case rollup instances are counted identically to detail instances, i.e. $skip=5 skips the first five instances, independently of whether some of them are rollup entities.~~
~~If a provider cannot satisfy a request using , it MUST respond with 501 Not Implemented and a human-readable error message.~~

~~3.17 Evaluating $apply as an Expand Option~~

~~The new system query option~~ $apply can be used as an expand or select option to inline the result of aggregating related entities~~.~~ or nested instances. The rules for evaluating $apply are applied in the context of the ~~expanded navigation, i.e.~~related collection of entities or the selected collection of instances, meaning this context defines the input set of the first transformation. Furthermore, $apply is evaluated first, and other expand or select options on the same (navigation) property are evaluated on the result of $apply.

Example ~~:~~46: products with aggregated sales~~:~~

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

results in

```
{
```

~~*results in*~~

```
~~{~~
```

```
"@~~odata.~~context":"$metadata#~~Poducts(Salees(Amount)",~~Products(Sales(Total))",
  "value": [
~~{~~
    { "ID": "P2", "Name": "Coffee", "Color": "Brown", "TaxRate": 0.06,
      "Sales": [ { ~~"@odata.id": null,~~"Total@type": "Decimal", "Total":   12 } ] },
~~{~~
    { "ID": "P3", "Name": "Paper",  "Color": "White", "TaxRate": 0.14,
      "Sales": [ { ~~"@odata.id": null,~~"Total@type": "Decimal", "Total":    8 } ] },
~~{~~
    { "ID": "P4", "Name": "Pencil", "Color": "Black", "TaxRate": 0.14,
      "Sales": [ { ~~"@odata.id": null,~~ "Total": null } ] },
~~{~~
    { "ID": "P1", "Name": "Sugar",  "Color": "White", "TaxRate": 0.06,
      "Sales": [ { ~~"@odata.id": null,~~"Total@type": "Decimal", "Total":    4 } ] }
~~]~~
```

```
~~}~~
```

~~3.18 ABNF for Extended URL Conventions~~

```
  ]
}
```

3.9 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 Instances~~

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

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

~~Dynamic properties introduced through an or with custom aggregates are represented as defined by the response format.~~

~~Aggregated instances are logically instances of the declared type of the collection identified by the resource path of the request. If the resource path identifies a collection of entities, the aggregated instances are also entities. These aggregated entities can be transient or persistent. Transient entities don't possess an edit link or read link, and in the JSON representation are marked with `"@odata.id": null`, see . 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 **[]**, sections 4.1 and 4.2 MUST be met.~~

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

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

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

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

~~5 Cross-Joins and Aggregation~~

4 Cross-Joins and Aggregation

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

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

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

Example ~~:~~47: if Sale~~s~~ 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 ~~~/$~~/service/$crossjoin(Products,Sales)

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

results in

```
{
```

~~results in~~

```
~~{~~
```

```
  "@~~odata.~~context": "$metadata#Collection(Edm.ComplexType)",
  "value": [
    { "Products": { "Name": "Paper" }, "Sales": { "Amount":
1 } },
    { "Products": { "Name": "Sugar" }, "Sales": { "Amount":
2 } },
    …
```

```
  ~~}~~ ...
  ]
}
```

Example ~~:~~48: using the $crossjoin resource for aggregate queries

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

results in

```
{
```

~~*results in*~~

```
~~{~~
```

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

The entity container may be annotated in the same way as entity sets to express which aggregate queries are supported, see section 5~~section~~ .

~~6 Vocabulary for Data Aggregation~~

5 Vocabulary for Data Aggregation

The following terms are defined in the vocabulary for data aggregation OData-VocAggr.

5.1 Aggregation Capabilities

~~6.1 Aggregation Capabilities~~

The term ApplySupported can be applied to an entity set, an entity type, or a collection if the target path of the annotation starts with an entity container (see example 50~~or to structured types and~~). It describes the aggregation capabilities of the ~~entity container or of collections of instances of the annotated structured types.~~annotated target. If present, it implies that instances of the annotated ~~structured type, or of structured types used in the annotated entity container,~~target 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:

- The Transformations collection lists all supported set transformations. Allowed values are the names of the standard transformations ~~, , , , , , , , , , ,~~introduced in sections 3 and ~~, or a~~6, and namespace-qualified names identifying a service-defined bindable function. If Transformations is omitted the server supports all transformations defined by this specification.
- The CustomAggregationMethods collection lists supported custom aggregation methods. Allowed values are namespace-qualified names identifying service-specific aggregation methods. If omitted, no custom aggregation methods are supported.
- Rollup specifies whether the service supports no rollup, only a single rollup hierarchy, or multiple rollup hierarchies in a groupby~~groupby~~ transformation. If omitted, multiple rollup hierarchies are supported.
- A non-empty GroupableProperties indicates that only the listed properties of the annotated target can be used in groupby.
- A non-empty AggregatableProperties indicates that only the listed properties of the annotated target can be used in aggregate~~PropertyRestrictions specifies whether all properties can be used in and . 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 or .~~

- , optionally restricted to the specified aggregation methods.

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

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

The term ApplySupportedDefaults can be applied to an entity container. It allows to specify default support for aggregation capabilities Transformations, CustomAggregationMethods and Rollup that propagate to all collection-valued resources in the container. Annotating a specific collection-valued resource with the term ApplySupported overrides the default support with the specified properties using PATCH semantics:

- Primitive or collection-valued properties specified in ApplySupported replace the corresponding properties specified in ApplySupportedDefaults.
- Complex-valued properties specified in ApplySupported override the corresponding properties specified in ApplySupportedDefaults using PATCH semantics recursively.
- Properties specified neither in ApplySupported nor in ApplySupportedDefault have their default value.

Example 49: an entity container with default support for everything defined in this specification

```
<EntityContainer Name="SalesData">
  <Annotation Term="Aggregation.ApplySupportedDefaults" />
  ...
</EntityContainer>
```

Example 50~~Property~~: Define aggregation support only for the products of a given category

```
<Annotations
Target="SalesModel.SalesData/Categories/Products">
```

~~6.1.1 Groupable Properties~~

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

~~6.1.2 Aggregatable Properties~~

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

~~6.1.3 Custom Aggregates~~

```
  <Annotation Term="Aggregation.ApplySupported">
    ...
  </Annotation>
</Annotations>
```

5.2 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, whether they are null, and which input values are used.

When applied to ~~a structured~~an entity set, an entity type, or a collection if the target path of the annotation starts with an entity container, the annotation specifies custom aggregates that are available for ~~collections of~~ its instances ~~of that structured type~~and for aggregated instances resulting from these instances. When applied to an entity container, the annotation specifies custom aggregates whose input set may span multiple entity sets within the container.

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

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

If the custom aggregate is associated with ~~a structured type~~an entity set, entity type, or collection, the value of the Qualifier attribute MAY be identical to the name of a declared property of the ~~structured type. In~~ instances in this ~~case~~set or collection. In these cases, the value of the annotation MUST have the same value as the Type attribute of the declared property. This is typically done when the custom aggregate is used as a default aggregate for that property. In this case~~,~~ the name refers to the custom aggregate within an aggregate expression without a with 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 children.

Example ~~:~~51: Sales forecasts are modeled as a custom aggregate of the Sale~~s~~ entity type because it belongs there. For the budget, there is no appropriate structured type, so it is modeled as a custom aggregate of the SalesData *entity container.*

```
<Annotations Target="SalesModel.SalesData/Sales">
  <Annotation Term="Aggregation.CustomAggregate"
Qualifier="Forecast"
              String="Edm.Decimal" />
</Annotations>
<Annotations Target="SalesModel.SalesData">
  <Annotation Term="Aggregation.CustomAggregate"
Qualifier="Budget"
              String="Edm.Decimal" />
</Annotations>
```

These custom aggregates can be used in the aggregate *transformation:*

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

and:

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

5.3 Context-Defining Properties

~~6.1.4 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 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.

5.4 Annotation Example

~~6.1.5 Example~~

Example ~~:~~52: This simplified Sales entity ~~type~~set has a single aggregatable property Amount whose context is defined by the Code property of the related Currency, and a custom aggregate Forecast with the same context. The Code property of Currenc~~y~~ies is groupable. All other properties are neither groupable nor aggregatable.

```
<EntityType Name="Currency">
  <Key>
    <PropertyRef Name="Code" />
  </Key>
  <Property Name="Code" Type="Edm.String~~">~~" />
```

```
~~    <Annotation Term="Aggregation.Groupable" />~~
~~  </Property>~~
```

```
  <Property Name="Name" Type="Edm.String">
    <Annotation Term="Core.IsLanguageDependent" />
  </Property>
</EntityType>

<EntityType Name="Sales">
  <Key>
    <PropertyRef Name="ID" />
  </Key>
  <Property Name="ID" Type="Edm.String" Nullable="false" />
  <Property Name="Amount" Type="Edm.Decimal"
Scale="variable">
```

```
~~    <Annotation Term="Aggregation.Aggregateable" />~~
```

```
    <Annotation
Term="Aggregation.ContextDefiningProperties">
      <Collection>
        <PropertyPath>Currency/Code</PropertyPath>
      </Collection>
    </Annotation>
  </Property>
  <NavigationProperty Name="Currency"
Type="SalesModel.Currency"
                      Nullable="false~~">~~" />
```

```
~~    <Annotation Term="Aggregation.Groupable" />~~
~~  </NavigationProperty>~~

~~  <Annotation Term="Aggregation.CustomAggregate" Qualifier="Forecast"~~
~~              String="Edm.Decimal">~~
~~    <Annotation Term="Aggregation.ContextDefiningProperties">~~
~~      <Collection>~~
~~        <PropertyPath>Currency/Code</PropertyPath>~~
~~      </Collection>~~
```

```
    ~~    </Annotation>~~
```

```
~~  </Annotation>~~
```

```
</EntityType>

<EntityContainer Name="SalesData">
  <EntitySet Name="Sales" EntityType="SalesModel.Sale">
    <Annotation Term="Aggregation.ApplySupported">
      <Record>
        <PropertyValue Property="~~PropertyRestrictions" Bool="true"~~
~~/>~~AggregatableProperties">
    ~~</~~      <Collection>
            <Record>
              <PropertyValue Property="Property"
PropertyPath="Amount" />
            </Record>
          </Collection>
        </PropertyValue>
        <PropertyValue Property="GroupableProperties">
          <Collection>
            <PropertyPath>Currency</PropertyPath>
          </Collection>
        </PropertyValue>
      </Record>
    </Annotation>
~~</~~
    <Annotation~~>~~    Term="Aggregation.CustomAggregate"
Qualifier="Forecast"
```

```
~~<~~                String="Edm.Decimal">
      <Annotation
Term="Aggregation.ContextDefiningProperties">
        <Collection>
          <PropertyPath>Currency/Code</PropertyPath>
        </Collection>
      </Annotation>
    </Annotation>
  </EntitySet~~ Name="Sales" EntityType="SalesModel.Sales" />~~>

  <EntitySet Name="Currencies"
EntityType="SalesModel.Currency~~" />~~">
    <Annotation Term="Aggregation.ApplySupported">
      <Record>
        <PropertyValue Property="GroupableProperties">
          <Collection>
            <PropertyPath>Code</PropertyPath>
          </Collection>
        </PropertyValue>
      </Record>
    </Annotation>
  </EntitySet>
</EntityContainer>
```

5.5 Hierarchies

~~6.2 Hierarchies~~

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

5.5.1 Leveled Hierarchy

~~6.2.1 Leveled Hierarchy~~

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

A leveled hierarchy can be defined for a collection of instances of an entity or complex type and 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~~paths from the entity or complex type where the term is applied to groupable properties representing the levels, starting with the root level (coarsest granularity) down to the lowest (finest-grained) level of the hierarchy.

The term LeveledHierarchy ~~can only~~MUST be applied ~~to entity types, and the applying Annotation element MUST specify the Qualifier attribute. The value of the Qualifier attribute~~ with a qualifier that can be used to reference the hierarchy in grouping with rollup.

5.5.2 Recursive Hierarchy

~~6.2.2 Recursive Hierarchy~~

A recursive hierarchy ~~organizes the values of~~ is defined on a collection of entities by

- determining which entities are part of the hierarchy and giving every such entity a single ~~groupable property as nodes of a tree structure. This structure does not need to be as uniform as a leveled hierarchy. It~~ primitive non-null value that uniquely identifies it within the hierarchy. These entities are called *nodes*, and the primitive value is called the *node identifier*, and
- associating with every node zero or more nodes from the same collection, called its *parent nodes*.

The recursive hierarchy is described ~~by a~~ in the model by an annotation of the entity type with the complex term RecursiveHierarchy with these properties:

- The NodeProperty ~~contains the~~MUST be a path ~~to the~~ with single-valued segments ending in a primitive property. This property holds the node identifier of ~~the node~~an entity that is a node in the hierarchy.

~~•~~ The ParentNavigationProperty ~~allows~~MUST be a collection-valued or nullable single-valued navigation ~~to the entity representing the parent node.~~

~~• The optional DistanceFromRootProperty contains the path to a property that contains the number of edges between the node and the root node.~~

- ~~The optional IsLeafProperty contains the path to a Boolean~~ property ~~that indicates whether the~~ path that addresses the entity type annotated with this term. It navigates from an entity that is a node ~~is a leaf of~~in the hierarchy to its parent nodes.

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~~MUST be applied with a qualifier, which is used to reference the hierarchy in transformations operating on recursive hierarchies, in grouping with rolluprecursive, and in hierarchy functions~~.~~. The same entity can serve as nodes in different recursive hierarchies, given different qualifiers.

A *root node* is a node without parent nodes. A recursive hierarchy can have one or more root nodes. A node is a *child node* of its parent nodes, a node without child nodes is a *leaf node*. Two nodes with a common parent node are *sibling nodes* and so are two root nodes.

The *descendants with maximum distance \(d≥1\)* of a node are its child nodes and, if \(d>1\), the descendants of these child nodes with maximum distance \(d-1\). The *descendants* are the descendants with maximum distance \(d=∞\). A node together with its descendants forms a *sub-hierarchy* of the hierarchy.

The *ancestors with maximum distance \(d≥1\)* of a node are its parent nodes and, if \(d>1\), the ancestors of these parent nodes with maximum distance \(d-1\). The *ancestors* are the ancestors with maximum distance \(d=∞\). The ParentNavigationProperty MUST be such that no node is an ancestor of itself, in other words: cycles are forbidden.

~~6.2.2.1~~ The term UpPath can be used in hierarchical result sets to associate with each instance one of its ancestors, one ancestor of that ancestor and so

on. This instance annotation is introduced in section 6.2.2~~**Hierarchy Filter Functions**~~

.

5.5.2.1 Hierarchy Functions

For testing the position of a given entity ~~instance~~ in a recursive hierarchy ~~annotated to the entity's type~~, the Aggregation vocabulary OData-VocAggr defines unbound functions ~~that can be applied to any entity in $filter expressions:~~. These have

- a parameter pair HierarchyNodes, HierarchyQualifier where HierarchyNodes is a collection and HierarchyQualifier is the qualifier of a RecursiveHierarchy annotation on its common entity type. The node identifiers in this collection define the recursive hierarchy.
- a parameter Node that contains the node identifier of the entity to be tested. Note that the test result depends only on this node identifier, not on any other property of the given entity
- additional parameters, depending on the type of test (see below)
- a Boolean return value for the outcome of the test.

The following functions are defined:

- isnode tests if the given entity is a node of the hierarchy.
- isroot tests if the given entity is a root node of the hierarchy.
- isdescendant tests if the given entity is a descendant with maximum distance MaxDistance of an ancestor node (whose node identifier is given in a parameter Ancestor), or equals the ancestor if IncludeSelf is true.
- isancestor tests if the given entity is an ancestor with maximum distance MaxDistance of a descendant node (whose node identifier is given in a parameter Descendant), or equals the descendant if IncludeSelf is true.
- issibling tests if the given entity and another entity (whose node identifier is given in a parameter Other) are sibling nodes.
- isleaf tests if the given entity is a leaf node.

Another function rollupnode is defined that can only be used in connection with rolluprecursive.

5.5.3 Hierarchy Examples

The hierarchy terms can be applied to the Example Data Model~~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.~~

~~6.2.3 Examples~~

~~Example :~~.

⚠ Example 53: *leveled hierarchies for products and time, and a recursive hierarchy for the sales organizations:*

```
<edmx:Edmx xmlns:edmx~~=""~~
="http://docs.oasis-open.org/odata/ns/edmx"
           Version="4.0">
  <edmx:Reference Uri="https://docs.oasis-
open.org/odata/odata-data-
    aggregation-
ext/v4.0/csd0~~1~~4/vocabularies/Org.OData.Aggregation.V1.xml">
    <edmx:Include Alias="Aggregation"
                  Namespace="Org.OData.Aggregation.V1" />
~~</edmx:Reference>~~
  </edmx:Reference>
  <edmx:DataServices>
    <Schema xmlns~~=""~~
="http://docs.oasis-open.org/odata/ns/edm"
            Alias="SalesModel"
Namespace="org.example.odata.salesservice">
      <Annotations Target="SalesModel.Product">
        <Annotation Term="Aggregation.LeveledHierarchy"
                    Qualifier="ProductHierarchy">
          <Collection>
            <PropertyPath>Category/Name</PropertyPath>
            <PropertyPath>Name</PropertyPath>
          </Collection>
        </Annotation>
      </Annotations>

      <Annotations Target="SalesModel.Time">
        <Annotation Term="Aggregation.LeveledHierarchy"
                    Qualifier="TimeHierarchy">
          <Collection>
            <PropertyPath>Year</PropertyPath>
            <PropertyPath>Quarter</PropertyPath>
            <PropertyPath>Month</PropertyPath>
          </Collection>
        </Annotation>
      </Annotations>
      <Annotations Target="SalesModel.SalesOrganization">
        <Annotation Term="Aggregation.RecursiveHierarchy"
                    Qualifier="SalesOrgHierarchy">
          <Record>
            <PropertyValue Property="NodeProperty"
                           PropertyPath="ID" />
            <PropertyValue
Property="ParentNavigationProperty"
                           PropertyPath="Superordinate" />
          </Record>
        </Annotation>
      </Annotations>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

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

Example ~~:~~54: requesting all organizations below EMEA

```
GET ~~~/~~/service/SalesOrganizations~~?~~
    ~~$~~?$filter=
   ~~$it/~~Aggregation.isdescendant(~~Hierarchy~~
  HierarchyNodes=$root/SalesOrganizations,
  HierarchyQualifier='SalesOrgHierarchy',
  Node=ID,
  Ancestor='EMEA')
```

results in

```
{
```

  "@~~*results in*~~

```
~~{~~
```

```
 ~~"@odata.~~context": "$metadata#SalesOrganizations",
  "value": [
    { "ID": "EMEA Central",      "Name": "EMEA Central" },
    { "ID": "Sales Netherlands", "Name": "Sales Netherlands"
},
    { "ID": "Sales Germany",     "Name": "Sales Germany" },
    { "ID": "EMEA South",        "Name": "EMEA South" },
  ~~…~~

    ...
    { "ID": "EMEA North",        "Name": "EMEA North" },
  ~~…~~
~~]~~
    ...
  ]
}
```

Example ~~:~~55: requesting just those organizations directly below EMEA

```
GET /service/SalesOrganizations?$filter=
         ~~$it/~~Aggregation.isdescendant(~~Hierarchy~~
  HierarchyNodes=$root/SalesOrganizations,
  HierarchyQualifier='SalesOrgHierarchy',
  Node=ID,
  Ancestor='EMEA',
  MaxDistance=1)
```

results in

```
{
```

  "@~~*results in*~~

```
~~{~~
```

```
~~  "@odata.~~context": "$metadata#SalesOrganizations",
  "value": [
    { "ID": "EMEA Central", "Name": "EMEA Central" },
    { "ID": "EMEA South",   "Name": "EMEA South" },
    { "ID": "EMEA North",   "Name": "EMEA North" },
  …
~~]~~
    ...
  ]
}
```

Example ~~:~~56: just the lowest-level organizations

```
GET /service/SalesOrganizations?$filter=
            ~~$it/~~Aggregation.isleaf(~~Hierarchy~~
  HierarchyNodes=$root/SalesOrganizations,
  HierarchyQualifier='SalesOrgHierarchy'~~)~~,
  Node=ID)
```

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 ~~:~~57: the lowest-level organizations including their superordinate's `ID`

```
GET /service/SalesOrganizations?$filter=
            ~~$it/~~Aggregation.isleaf(~~Hierarchy~~
  HierarchyNodes=$root/SalesOrganizations,
  HierarchyQualifier='SalesOrgHierarchy'~~)~~
           ,
  Node=ID)
&$expand=Superordinate($select=ID)
```

results in

```
{
```

~~results in~~

```
~~{~~
```

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

Example ~~: retrieving~~58: the sales IDs involving sales organizations from EMEA ~~can be requested by~~

```
GET /service/Sales?$select=ID&$filter=
  ~~SalesOrganization/~~Aggregation.isdescendant(~~Hierarchy~~
  HierarchyNodes=$root/SalesOrganizations,
  HierarchyQualifier='SalesOrgHierarchy',
  Node=SalesOrganization/ID,
  Ancestor='EMEA')
```

results in

```
{
```

~~results in~~

```
~~{~~
```

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

Further examples for recursive hierarchies using transformations operating on the hierarchy structure are provided in section 7.9.

5.6 Functions on Aggregated Entities~~}~~

~~6.3 Actions and Functions on Aggregated Entities~~

~~Bound actions and~~

Service-defined bound functions ~~may or may not be applicable to aggregated entities. By default such bindings are not applicable to aggregated entities. Actions or functions~~that serve as set transformations MAY be annotated with the term AvailableOnAggregates to indicate that they 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 ~~:~~59: assume the product is an implicit input for a function ~~bindable~~bound to a collection of Sales*, then aggregating away the product makes this function inapplicable.*

6 Hierarchical Transformations

The transformations and the rolluprecursive operator defined in this section are called hierarchical, because they make use of a recursive hierarchy and are defined in terms of hierarchy functions introduced in the previous section.

The transformations ancestors and descendants do not define an order on the output set. An order can be imposed by a subsequent orderby or traverse transformation or a $orderby. The output set of traverse is in preorder or postorder, and grouping with rolluprecursive orders its output set in analogy with simple grouping.

The algorithmic descriptions of the transformations make use of a *union* of collections, this is defined as an unordered collection containing the items from all these collections and from which duplicates have been removed.

The notation \(u[t]\) is used to denote the value of a property \(t\), possibly preceded by a type-cast segment, in an instance \(u\). It is also used to denote the value of a single-valued data aggregation path \(t\), evaluated relative to \(u\). The value of a collection-valued data aggregation path is denoted in the \(\Gamma\) notation by \(γ(u,t)\).

The notations introduced here are used throughout the following subsections.

6.1 Common Parameters for Hierarchical Transformations

The parameter lists defined in the following subsections have three mandatory parameters in common.

The recursive hierarchy is defined by a parameter pair \((H,Q)\), where \(H\) and \(Q\) MUST be specified as the first and second parameter. Here, \(H\) MUST be an expression of type Collection(Edm.EntityType) starting with $root that has no multiple occurrences of the same entity. \(H\) identifies the collection of node entities forming a recursive hierarchy based on an annotation of their common entity type with term RecursiveHierarchy with a Qualifier attribute whose value MUST be provided in \(Q\). The property paths referenced by NodeProperty and ParentNavigationProperty in the RecursiveHierarchy annotation must be evaluable for the nodes in the recursive hierarchy, otherwise the service MUST reject the request. The NodeProperty is denoted by \(q\) in this section.

The third parameter MUST be a data aggregation path \(p\) with single- or collection-valued segments whose last segment MUST be a primitive property. The node identifier(s) of an instance \(u\) in the input set are the primitive values in \(γ(u,p)\), they are reached via \(p\) starting from \(u\). Let \(p=p_1/…/p_k/r\) with \(k≥0\) be the concatenation where each sub-path \(p_1,…,p_k\) consists of a collection-valued segment that is preceded by zero or more single-valued segments, and either \(r\) consists of one or more single-valued segments or \(k≥1\) and \({}/r\) is absent. Each segment can be prefixed with a type cast.

Some parameter lists allow as optional fourth or fifth parameter a non-empty sequence \(S\) of transformations. The transformation sequence \(S\) will be applied to the node collection \(H\). It MUST consist of transformations listed in section 3.3 or section 6.2 or service-defined bound functions whose output set is a subset of their input set.

6.2 Hierarchical Transformations Producing a Subset

These transformations produce an output set that consists of certain instances from their input set, possibly with repetitions or in a different order.

6.2.1 Transformations ancestors and descendants

In the simple case, the ancestors transformation takes an input set consisting of instances that belong to a recursive hierarchy \((H,Q)\). It determines a subset \(A\) of the input set and then determines the set of ancestors of \(A\) that were already contained in the input set. Its output set is the ancestors set, optionally including \(A\).

In the more complex case, the instances in the input set are instead related to nodes in a recursive hierarchy. Then the ancestors transformation determines a subset \(A\) of the input set consisting of instances that are related to certain nodes in the hierarchy, called start nodes. The ancestors of these start nodes are then determined, and the output set consists of instances of the input set that are related to the ancestors, or optionally to the start nodes.

The descendants transformation works analogously, but with descendants.

\(H\), \(Q\) and \(p\) are the first three parameters defined above.

The fourth parameter is a transformation sequence \(T\) composed of transformations listed section 3.3 or section 6.2.1 and of service-defined bound functions whose output set is a subset of their input set. \(A\) is the output set of this sequence applied to the input set.

The fifth parameter \(d\) is optional and takes an integer greater than or equal to 1 that specifies the maximum distance between start nodes and ancestors or descendants to be considered. An optional final keep start parameter drives the optional inclusion of the subset or start nodes.

The output set of the transformation \({\tt ancestors}(H,Q,p,T,d,{\tt keep\ start})\) or \({\tt descendants}(H,Q,p,T,d,{\tt keep\ start})\) is defined as the union of the output sets of transformations \(F(u)\) applied to the input set for all \(u\) in \(A\). For a given instance \(u\), the transformation \(F(u)\) determines all instances of the input set whose node identifier is an ancestor or descendant of the node identifier of \(u\):

If \(p\) contains only single-valued segments, then, for ancestors, \[\matrix{
F(u)={\tt filter}(\hbox{\tt Aggregation.isancestor}(\hfill\\ \quad {\tt
HierarchyNodes}=H,\;{\tt HierarchyQualifier}=\hbox{\tt{'Q'}},\hfill\\ \quad

{\tt Node}=p,\;{\tt Descendant}=u[p],\;{\tt MaxDistance}=d,\;{\tt IncludeSelf}={\tt true}))\hfill }\] or, for descendants, \[\matrix{ F(u)={\tt filter}(\hbox{\tt Aggregation.isdescendant}(\hfill\\ \quad {\tt HierarchyNodes}=H,\;{\tt HierarchyQualifier}=\hbox{\tt{'Q'}},\hfill\\ \quad {\tt Node}=p,\;{\tt Ancestor}=u[p],\;{\tt MaxDistance}=d,\;{\tt IncludeSelf}={\tt true})).\hfill }\]

Otherwise \(p=p_1/…/p_k/r\) with \(k≥1\), in this case the output set of the transformation \(F(u)\) is defined as the union of the output sets of transformations \(G(n)\) applied to the input set for all \(n\) in \(γ(u,p)\). The output set of \(G(n)\) consists of the instances of the input set whose node identifier is an ancestor or descendant of the node identifier \(n\):

For ancestors, \[\matrix{ G(n)={\tt filter}(\hfill\\ \hskip1pc p_1/{\tt any}(y_1:\hfill\\ \hskip2pc y_1/p_2/{\tt any}(y_2:\hfill\\ \hskip3pc ⋱\hfill\\ \hskip4pc y_{k-1}/p_k/{\tt any}(y_k:\hfill\\ \hskip5pc \hbox{\tt Aggregation.isancestor}(\hfill\\ \hskip6pc {\tt HierarchyNodes}=H,\;{\tt HierarchyQualifier}=\hbox{\tt{'Q'}},\hfill\\ \hskip6pc {\tt Node}=y_k/r,\;{\tt Descendant}=n,\;{\tt MaxDistance}=d,\;{\tt IncludeSelf}={\tt true}\hfill\\ \hskip5pc)\hfill\\ \hskip4pc)\hfill\\ \hskip3pc ⋰\hfill\\ \hskip2pc)\hfill\\ \hskip1pc)\hfill\\ )\hfill }\] or, for descendants, \[\matrix{ G(n)={\tt filter}(\hfill\\ \hskip1pc p_1/{\tt any}(y_1:\hfill\\ \hskip2pc y_1/p_2/{\tt any}(y_2:\hfill\\ \hskip3pc ⋱\hfill\\ \hskip4pc y_{k-1}/p_k/{\tt any}(y_k:\hfill\\ \hskip5pc \hbox{\tt Aggregation.isdescendant}(\hfill\\ \hskip6pc {\tt HierarchyNodes}=H,\;{\tt HierarchyQualifier}=\hbox{\tt{'Q'}},\hfill\\ \hskip6pc {\tt Node}=y_k/r,\;{\tt Ancestor}=n,\;{\tt MaxDistance}=d,\;{\tt IncludeSelf}={\tt true}\hfill\\ \hskip5pc)\hfill\\ \hskip4pc)\hfill\\ \hskip3pc ⋰\hfill\\ \hskip2pc)\hfill\\ \hskip1pc)\hfill\\ )\hfill }\] where \(y_1,…,y_k\) denote lambdaVariableExprs as defined in OData-ABNF and \({}/r\) may be absent.

If parameter \(d\) is absent, the parameter \({\tt MaxDistance}=d\) is omitted. If keep start is absent, the parameter \({\tt IncludeSelf}={\tt true}\) is omitted.

Since the output set of ancestors is constructed as a union, no instance from the input set will occur more than once in it, even if, for example, a sale is related to both a sales organization and one of its ancestor organizations. For descendants, analogously.

Example 60: Request based on the SalesOrgHierarchy defined in Hierarchy Examples, with Superordinate/$ref expanded to illustrate the hierarchy relation

```
GET /service/SalesOrganizations?$apply=
    ancestors($root/SalesOrganizations,SalesOrgHierarchy,ID,
              filter(contains(Name,'East') or
contains(Name,'Central')))
```

```
  &$expand=Superordinate/$ref
```

results in

```
{
  "@context": "$metadata#SalesOrganizations",
  "value": [
    { "ID": "EMEA",  "Name": "EMEA",
      "Superordinate": { "@id":
"SalesOrganizations('Sales')" } },
    { "ID": "US",    "Name": "US",
      "Superordinate": { "@id":
"SalesOrganizations('Sales')" } },
    { "ID": "Sales", "Name": "Sales",
      "Superordinate": null }
  ]
}
```

Example 61: Request based on the SalesOrgHierarchy defined in Hierarchy Examples, with Superordinate/$ref expanded to illustrate the hierarchy relation

```
GET /service/SalesOrganizations?$apply=

descendants($root/SalesOrganizations,SalesOrgHierarchy,ID,
                filter(Name eq 'US'),keep start)
  &$expand=Superordinate/$ref
```

results in

```
{
```

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

~~{~~

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

```
~~  "@odata.readLink": "http://.../aggregated-stuff2143248437259843",~~
~~"#Model.ColAction": {~~
~~    "title": "Do something on this collection",~~
~~    "target": "http://.../aggregated-stuff2143248437259843/Model.ColAction"~~
~~  },~~
```

```
"value": [
    { "ID": "US West", "Name": "US West",
      "Superordinate": { "@id": "SalesOrganizations('US')" }
},
    { "ID": "US",      "Name": "US",
      "Superordinate": { "@id":
"SalesOrganizations('Sales')" } },
    { "ID": "US East", "Name": "US East",
      "Superordinate": { "@id": "SalesOrganizations('US')" }
}
  ]
}
```

⚠ Example 62: Input set and recursive hierarchy from two different entity sets

```
GET /service/Sales?$apply=
    ancestors($root/SalesOrganizations,
              SalesOrgHierarchy,
              SalesOrganization/ID,
              filter(contains(SalesOrganization/Name,'East')
                  or
contains(SalesOrganization/Name,'Central')),
              keep start)
```

results in

```
{
  "@context": "$metadata#Sales",
  "value": [
    { "ID": "4", "Amount": 8,
      "SalesOrganization": { "ID": "US East",      "Name":
"US East" } },
    { "ID": "5", "Amount": 4,
      "SalesOrganization": { "ID": "US East",      "Name":
"US East" } },
    { "ID": "6", "Amount": 2,
      "SalesOrganization": { "ID": "EMEA Central", "Name":
"EMEA Central" } },
    { "ID": "7", "Amount": 1,
      "SalesOrganization": { "ID": "EMEA Central", "Name":
"EMEA Central" } },
    { "ID": "8", "Amount": 2,
      "SalesOrganization": { "ID": "EMEA Central", "Name":
"EMEA Central" } }
  ]
}
```

6.2.2 Transformation traverse

The traverse transformation returns instances of the input set that are or are related to nodes of a given recursive hierarchy in a specified tree order.

\(H\), \(Q\) and \(p\) are the first three parameters defined above.

The fourth parameter h of the `traverse` transformation is either `preorder` or `postorder`. S is an optional fifth parameter as defined above. Let H' be the output set of the transformation sequence S applied to H, or let H' be the collection of root nodes in the recursive hierarchy (H,Q) if S is not specified. Nodes in H' are called start nodes in this subsection (see example 117).

All following parameters are optional and form a list o of expressions that could also be passed as a `$orderby` system query option. If o is present, the transformation stable-sorts H' by o.

The instances in the input set are related to one node (if p is single-valued) or multiple nodes (if p is collection-valued) in the recursive hierarchy. Given a node x, denote by $\hat F(x)$ the collection of all instances in the input set that are related to x; these collections can overlap. For each u in $\hat F(x)$, the output set contains one instance that comprises the properties of u and additional properties that identify the node x. These additional properties are independent of u and are bundled into an instance called $σ(x)$. For example, if a sale u is related to two sales organizations and hence contained in both $\hat F(x_1)$ and $\hat F(x_2)$, the output set will contain two instances $(u,σ(x_1))$ and $(u,σ(x_2))$ and $σ(x_i)$ contributes a navigation property `SalesOrganization`.

A transformation $F(x)$ is defined below such that $\hat F(x)$ is the output set of $F(x)$ applied to the input set of the `traverse` transformation.

Given a node x, the formulas below contain the transformation $\Pi_G(σ(x))$ in order to inject the properties of $σ(x)$ into the instances in $\hat F(x)$; this uses the function Π_G that is defined in the simple grouping section. Further, G is a list of data aggregation paths that shall be present in the output set, and $σ$ is a function that maps each hierarchy node x to an instance of the input type containing the paths from G. As a consequence of the following definitions, only single-valued properties and "final segments from G" are nested into $σ(x)$, therefore the behavior of $\Pi_G(σ(x))$ is well-defined.

The definition of $σ(x)$ makes use of a function $a(ε,t,x)$, which returns a sparsely populated instance u in which only the path t has a value, namely $u[t]=x$.

Three cases are distinguished:

1. *Case where the recursive hierarchy is defined on the input set*
   This case applies if the paths p and q are equal. Let $σ(x)=x$ and let G be a list containing all structural and navigation

properties of the entity type of H.
In this case $\Pi_G(σ(x))$ injects all properties of x into the instances of the output set. (See example 65.)

2. *Case where the recursive hierarchy is defined on the related entity type addressed by a navigation property path*
   This case applies if p' is a non-empty navigation property path and p'' an optional type-cast segment such that p equals the concatenated path $p'/p''/q$. Let $σ(x)=a(ε,p'/p'',x)$ and let $G=(p')$.
   In this case $\Pi_G(σ(x))$ injects the whole related entity x into the instances of the output set. The navigation property path p' is expanded by default. (See example 66.)
3. *Case where the recursive hierarchy is related to the input set only through equality of node identifiers, not through navigation*
   If neither case 1 nor case 2 applies, let $σ(x)=a(ε,p,x[q])$ and let $G=(p)$.
   In this case $\Pi_G(σ(x))$ injects only the node identifier of x into the instances of the output set.

Here paths are considered equal if their non-type-cast segments refer to the same model elements when evaluated relative to the input set (see example 68).

The function $a(u,t,x)$ takes an instance, a path and another instance as arguments and is defined recursively as follows:

1. If u equals the special symbol $ε$, set u to a new instance of the input type without properties and without entity id.
2. If t contains only one segment other than a type cast, let $t_1=t$, and let $x'=x$, then go to step 6.
3. Otherwise, let t_1 be the first property segment in t, possibly together with a preceding type-cast segment, let t_2 be any type-cast segment that immediately follows, and let t_3 be the remainder such that t equals the concatenated path $t_1/t_2/t_3$ where ${}/t_2$ may be absent.
4. Let u' be an instance of the type of t_1/t_2 without properties and without entity id.
5. Let $x'=a(u',t_3,x)$.
6. If t_1 is single-valued, let $u[t_1]=x'$.
7. If t_1 is collection-valued, let $u[t_1]$ be a collection consisting of one item x'.
8. Return u.

(See example 112.)

6.2.2.1 Standard Case of traverse

The algorithm is first given for the standard case where RecursiveHierarchy/ParentNavigationProperty is single-valued and the optional parameter \(S\) is not specified. In this standard case, start nodes are root nodes and \(σ(x)\) is computed exactly once for every node \(x\), as part of the recursive formula for \(R(x)\) given below. The general case follows later.

Let \(r_1,…,r_n\) be a sequence of the start nodes in \(H'\) preserving the order of \(H'\) stable-sorted by \(o\). Then the transformation \({\tt traverse}(H,Q,p,h,o)\) is defined as equivalent to \[{\tt concat}(R(r_1),…,R(r_n)).\]

\(R(x)\) is a transformation producing the specified tree order for a sub-hierarchy of \(H\) with root node \(x\). Let \(c_1,…,c_m\) with \(m≥0\) be an order-preserving sequence of the children of \(x\) in \((H,Q)\). The *recursive formula for \(R(x)\)* is as follows:

If \(h={\tt preorder}\), then \[R(x)={\tt concat}(F(x)/\Pi_G(σ(x)),R(c_1),…,R(c_m)).\]

If \(h={\tt postorder}\), then \[R(x)={\tt concat}(R(c_1),…,R(c_m),F(x)/\Pi_G(σ(x))).\]

The absence of cycles guarantees that the recursion terminates.

\(F(x)\) is a transformation that determines for the specified node \(x\) the instances of the input set having the same node identifier as \(x\).

If \(p\) contains only single-valued segments, then \[F(x)={\tt filter}(p{\tt\ eq\ }x[q]).\]

Otherwise \(p=p_1/…/p_k/r\) with \(k≥1\) and \[\matrix{ F(x)={\tt filter}(\hfill\\ \hskip1pc p_1/{\tt any}(y_1:\hfill\\ \hskip2pc y_1/p_2/{\tt any}(y_2:\hfill\\ \hskip3pc ⋱\hfill\\ \hskip4pc y_{k-1}/p_k/{\tt any}(y_k:\hfill\\ \hskip5pc y_k/r{\tt\ eq\ }x[q]\hfill\\ \hskip4pc)\hfill\\ \hskip3pc ⋰\hfill\\ \hskip2pc)\hfill\\ \hskip1pc)\hfill\\ )\hfill }\] where \(y_1,…,y_k\) denote lambdaVariableExprs and \({}/r\) may be absent.

Example 63: Based on the SalesOrgHierarchy defined in Hierarchy Examples

```
GET /service/SalesOrganizations?$apply=

descendants($root/SalesOrganizations,SalesOrgHierarchy,ID,
                   Name eq 'US',keep start)
```

```
/ancestors($root/SalesOrganizations,SalesOrgHierarchy,ID,
                contains(Name,'East'),keep start)

/traverse($root/SalesOrganizations,SalesOrgHierarchy,ID,preo
rder)
  &$expand=Superordinate/$ref
```

results in

```
{
  "@context": "$metadata#SalesOrganizations",
  "value": [
    { "ID": "US",       "Name": "US",
      "Superordinate": { "@id":
"SalesOrganizations('Sales')" } },
    { "ID": "US East", "Name": "US East",
      "Superordinate": { "@id": "SalesOrganizations('US')" }
}
  ]
}
```

6.2.2.2 General Case of traverse

In the general case, the recursive algorithm can reach a node \(x\) multiple times, via different parents or ancestors, or because \(x\) is a start node and a descendant of another start node. Then the algorithm computes \(R(x)\) and hence \(σ(x)\) multiple times. In order to distinguish these computation results, information about the ancestors up to the start node is injected into each \(σ(x)\) by annotating \(x\) differently before each \(σ(x)\) is computed. On the other hand, certain nodes can be unreachable from any start node, these are called orphans of the traversal (see example 117).

More precisely, in the general case every node \(y\) is annotated with the term UpPath from the Aggregation vocabulary OData-VocAggr. The annotation has \(Q\) as qualifier and the annotation value is a collection of string values of node identifiers. The first member of that collection is the node identifier of the parent node \(x\) such that \(R(y)\) appears on the right-hand side of the recursive formula for \(R(x)\). The following members are the members of the Aggregation.UpPath collection of \(x\). Every instance in the output set of traverse is related to one node with Aggregation.UpPath annotation. Start nodes appear annotated with an empty collection.

⚠ Example 64: A sales organization Atlantis with two parents US and EMEA would occur twice in the result of a traverse transformation:

```
GET /service/SalesOrganizations?$apply=
traverse($root/SalesOrganizations,MultiParentHierarchy,ID,pr
eorder)
```

results in

```
{
  "@context": "$metadata#SalesOrganizations",
  "value": [
    ...
    { "ID": "Atlantis", "Name": "Atlantis",
      "@Aggregation.UpPath#MultiParentHierarchy":
        [ "US", "Sales" ] },
    { "ID": "AtlantisChild", "Name": "Child of Atlantis",
      "@Aggregation.UpPath#MultiParentHierarchy":
         [ "Atlantis", "US", "Sales" ] },
    ...
    { "ID": "Atlantis", "Name": "Atlantis",
      "@Aggregation.UpPath#MultiParentHierarchy":
        [ "EMEA", "Sales" ] },
    { "ID": "AtlantisChild", "Name": "Child of Atlantis",
      "@Aggregation.UpPath#MultiParentHierarchy":
        [ "Atlantis", "EMEA", "Sales" ] },
    ...
  ]
}
```

Given a start node \(x\), let \(ρ_0(x)\) be the node \(x\) with the annotation \(ρ_0(x)/@\hbox{\tt Aggregation.UpPath}\#Q=[]\) set to an empty collection.

Given a node \(x\) annotated with \(x/@\hbox{\tt Aggregation.UpPath}\#Q=[x_1,…,x_d]\), where \(d≥0\), and given a child \(y\) of \(x\), let \(ρ(y,x)\) be the node \(y\) with the annotation \[ρ(y,x)/@\hbox{\tt Aggregation.UpPath}\#Q=[{\tt cast}(x[q],\hbox{\tt Edm.String}),x_1,…,x_d].\]

Like structural and navigation properties, these instance annotations are considered part of the node \(x\) and are copied over to \(σ(x)\). For them to be included in the transformation \(\Pi_G(σ(x))\), an additional step is inserted between steps 2 and 3 of the function \(a_G(u,s,p)\) as defined in the simple grouping section:

- If \(s\) is annotated with Aggregation.UpPath and qualifier \(Q\), copy this annotation from \(s\) to \(u\).

Recall that instance annotations never appear in data aggregation paths or aggregatable expressions. They are not considered when determining whether instances of structured types are the same, they do not cause conflicting representations and are absent from merged representations.

Let \(r_1,…,r_n\) be the start nodes in \(H'\) as above, then the transformation \({\tt traverse}(H,Q,p,h,S,o)\) is defined as equivalent to \[{\tt concat}(R(ρ_0(r_1)),…,R(ρ_0(r_n))\] where the function \(R(x)\) takes as argument a node with Aggregation.UpPath annotation. With \(F(x)\) and \(c_1,…,c_m\) as above, if \(h={\tt preorder}\), then \[R(x)={\tt concat}(F(x)/\Pi_G(σ(x)),R(ρ(c_1,x)),…,R(ρ(c_m,x))),\] and if \(h={\tt postorder}\), then \[R(x)={\tt concat}(R(ρ(c_1,x)),…,R(ρ(c_m,x)),F(x)/\Pi_G(σ(x))).\]

The absence of cycles guarantees that the recursion terminates.

In the general case, servers MUST include the Aggregation.UpPath annotations in the result of $apply but MAY omit them if RecursiveHierarchy/ParentNavigationProperty is single-valued and all start nodes are root nodes.

If RecursiveHierarchy/ParentNavigationProperty is collection-valued but the parent collection never contains more than one parent and the optional parameter \(S\) is not specified, then the result is effectively like in the standard case, except for the presence of the Aggregation.UpPath annotations.

6.3 Grouping with rolluprecursive

Recall that simple grouping partitions the input set and applies a transformation sequence to each partition. By contrast, grouping with rolluprecursive, informally speaking, transforms the input set into overlapping portions (like "US" and "US East"), one for each node \(x\) of a recursive hierarchy. The transformation \(F(x)\), defined below, outputs the portion whose node identifiers are among the descendants of \(x\) (including \(x\) itself). A transformation sequence is then applied to each portion, and they are made distinguishable in the output set through injection of information about the node \(x\), which is achieved through the transformation \(\Pi_G(σ(x))\) defined in the traverse section.

As defined above, \(H\), \(Q\) and \(p\) are the first three parameters of rolluprecursive, \(S\) is an optional fourth parameter. Let \(H'\) be the output set of the transformation sequence \(S\) applied to \(H\), or \(H'=H\) if \(S\) is not specified.

Navigation properties specified in \(p\) are expanded by default.

Let \(T\) be a transformation sequence, \(P_1\) stand in for zero or more property paths and \(P_2\) for zero or more rollup or rolluprecursive operators or property paths. The transformation \({\tt groupby}((P_1,{\tt rolluprecursive}(H,Q,p,S),P_2),T)\) is computed by the following algorithm, which invokes itself recursively if the number of rolluprecursive operators in the first argument of the groupby transformation, which is called \(M\), is greater than one. Let \(N\) be the recursion depth of the algorithm, starting with 1.

The rolluprecursive algorithm:

A property \(χ_N\) appears in the algorithm, but is not present in the output set. It is explained later (see example 66). \(Z_N\) is a transformation whose output set is its input set with property \(χ_N\) removed.

Let \(x_1,…,x_n\) be the nodes in \(H'\), possibly with repetitions. If the optional transformation sequence \(S\) ends with a traverse transformation, as in example 118, the sequence \(x_1,…,x_n\) MUST have the preorder or postorder established by that traversal, and the transformation \({\tt groupby}((P_1,{\tt rolluprecursive}(H,Q,p,S),P_2),T)\) is defined as equivalent to \[{\tt concat}(R(x_1),…,R(x_n)).\]

Otherwise, if \(S\) is not specified or does not end with a traverse transformation, the output set of the transformation \({\tt groupby}((P_1,{\tt rolluprecursive}(H,Q,p,S),P_2),T)\) is the concatenation of \(R(x_1),…,R(x_n)\). The order of occurrences from the same \(R(x_i)\) remains the same, and no order is defined between occurrences from different \(R(x_i)\) and \(R(x_j)\).

\(R(x)\) is a transformation that processes the entire sub-hierarchy rooted at \(x\), which is the output set of \(F(x)\). The output set of \(R(x)\) is a collection of aggregated instances for all rollup results.

If at least one of \(P_1\) or \(P_2\) is non-empty, then \[R(x)=F(x)/{\tt compute}(x{\tt\ as\ }χ_N)/{\tt groupby}((P_1,P_2),T/Z_N/\Pi_G(σ(x))).\]

The property \(χ_N=x\) is present during the evaluation of \(T\), but not afterwards. If \(P_2\) contains a rolluprecursive operator, the evaluation of the formula involves a recursive invocation (with \(N\) increased by 1) of the rolluprecursive algorithm.

Otherwise if \(P_1\) and \(P_2\) are empty, then \[R(x)=F(x)/{\tt compute}(x{\tt\ as\ }χ_N)/T/Z_N/\Pi_G(σ(x)).\]

\(F(x)\) is defined as follows: If \(p\) contains only single-valued segments, then \[\matrix{ F(x)={\tt filter}(\hbox{\tt Aggregation.isdescendant}(\hfill\\ \quad {\tt HierarchyNodes}=H,\;{\tt HierarchyQualifier}=\hbox{\tt{'Q'}},\hfill\\ \quad {\tt Node}=p,\;{\tt Ancestor}=x[q],\;{\tt IncludeSelf}={\tt true})).\hfill }\]

Otherwise \(p=p_1/…/p_k/r\) with \(k≥1\) and \[\matrix{ F(x)={\tt filter}(\hfill\\ \hskip1pc p_1/{\tt any}(y_1:\hfill\\ \hskip2pc y_1/p_2/{\tt any}(y_2:\hfill\\ \hskip3pc ⋱\hfill\\ \hskip4pc y_{k-1}/p_k/{\tt any}(y_k:\hfill\\ \hskip5pc \hbox{\tt Aggregation.isdescendant}(\hfill\\ \hskip6pc {\tt HierarchyNodes}=H,\;{\tt HierarchyQualifier}=\hbox{\tt{'Q'}},\hfill\\ \hskip6pc {\tt Node}=y_k/r,\;{\tt Ancestor}=x[q],\;{\tt IncludeSelf}={\tt true}\hfill\\ \hskip5pc)\hfill\\ \hskip4pc)\hfill\\ \hskip3pc ⋰\hfill\\ \hskip2pc)\hfill\\ \hskip1pc)\hfill\\ )\hfill }\] where \(y_1,…,y_k\) denote lambdaVariableExprs and \({}/r\) may be absent. (See example 113 for a case with \(k=1\).)

Informatively speaking, the effect of the algorithm can be summarized as follows: If \(M≥1\) and \(\hat F_N(x)\) denotes the collection of all instances that are related to a node \(x\) as determined by \(F(x)\) in the recursive hierarchy of the \(N\)-th rolluprecursive operator, then \(T\) is applied to each of the intersections of \(\hat F_1(χ_1),…,\hat F_M(χ_M)\), as \(χ_N\) runs over all nodes of the \(N\)-th recursive hierarchy for \(1≤N≤M\). Into the instances of the resulting output sets the \(\Pi_G\) transformations inject information about the nodes \(χ_1,…,χ_M\).

Example 65: Total number of sub-organizations for all organizations in the hierarchy defined in Hierarchy Examples with \(p=q={\tt ID}\) (case 1 of the definition of \(σ(x)\)). In this case \(\Pi_G(σ(x))\) writes back the entire node into the output set of \(T\). aggregates must have an alias to avoid overwriting by a property of the node with the same name.

```
GET /service/SalesOrganizations?$apply=
    groupby((rolluprecursive(

$root/SalesOrganizations,SalesOrgHierarchy,ID)),
             aggregate($count as OrgCnt)/compute(OrgCnt sub
1 as SubOrgCnt))
  &$select=ID,Name,SubOrgCnt
  &$expand=Superordinate($select=ID)
```

results in

```
{
  "@context":

"$metadata#SalesOrganizations(ID,Name,SubOrgCnt,Superordinat
e(ID))",
  "value": [
    { "ID": "US West",       "Name": "US West",
      "SubOrgCount": 0, "Superordinate": { "ID": "US" } },
    { "ID": "US East",       "Name": "US East",
      "SubOrgCount": 0, "Superordinate": { "ID": "US" } },
    { "ID": "US",            "Name": "US",
      "SubOrgCount": 2, "Superordinate": { "ID": "Sales" }
},
    { "ID": "EMEA Central", "Name": "EMEA Central",
      "SubOrgCount": 0, "Superordinate": { "ID": "EMEA" } },
    { "ID": "EMEA",         "Name": "EMEA",
      "SubOrgCount": 1, "Superordinate": { "ID": "Sales" }
},
    { "ID": "Sales",        "Name": "Sales",
      "SubOrgCount": 5, "Superordinate": null }
  ]
}
```

The value of the property \(χ_N\) in the rolluprecursive algorithm is the node \(x\) at recursion level \(N\). In a common expression, \(χ_N\) cannot be accessed by its name, but can only be read as the return value of the unbound function \({\tt rollupnode}({\tt Position}=N)\) defined in the Aggregation vocabulary OData-VocAggr, with \(1≤N≤M\), and only during the application of the transformation sequence \(T\) in the formula for \(R(x)\) above (the function is undefined otherwise). If \(N=1\), the Position parameter can be omitted.

⚠ Example 66: Total sales amounts per organization, both including and excluding sub-organizations, in the US sub-hierarchy defined in Hierarchy Examples with \(p=p'/q={\tt SalesOrganization}/{\tt ID}\) and \(p'={\tt SalesOrganization}\) (case 2 of the definition of \(σ(x)\)). The Boolean expression \(p'\hbox{\tt\ eq Aggregation.rollupnode}()\) is true for sales in the organization for which the aggregate is computed, but not for sales in sub-organizations.

```
GET /service/Sales?$apply=groupby(
    (rolluprecursive(
      $root/SalesOrganizations,
      SalesOrgHierarchy,
      SalesOrganization/ID,
      descendants($root/SalesOrganizations,
                  SalesOrgHierarchy,
                  ID, filter(ID eq 'US'), keep start))),
    compute(case(SalesOrganization eq
Aggregation.rollupnode():Amount)
            as AmountExcl)
    /aggregate(Amount with sum as TotalAmountIncl,
               AmountExcl with sum as TotalAmountExcl))
```

results in

```
{
  "@context": "$metadata#Sales(SalesOrganization(),

TotalAmountIncl,TotalAmountExcl)",
  "value": [
    { "SalesOrganization": { "ID": "US West", "Name": "US
West" },
      "TotalAmountIncl@type": "Decimal", "TotalAmountIncl":
7,
      "TotalAmountExcl@type": "Decimal" ,"TotalAmountExcl":
7 },
    { "SalesOrganization": { "ID": "US",      "Name": "US"
},
      "TotalAmountIncl@type": "Decimal", "TotalAmountIncl":
19,
      "TotalAmountExcl": null },
    { "SalesOrganization": { "ID": "US East", "Name": "US
East" },
      "TotalAmountIncl@type": "Decimal", "TotalAmountIncl":
12,
      "TotalAmountExcl@type": "Decimal", "TotalAmountExcl":
12 }
  ]
}
```

⚠ Example 67: When requesting a sub-hierarchy consisting of the US East sales organization and its ancestors, the total sales amounts can either include the descendants outside this sub-hierarchy ("actual totals") or can exclude them ("visual totals").

Actual totals are computed when rolluprecursive is restricted to the sub-hierarchy by setting the optional parameter \(S\) to an ancestors transformation:

```
GET /service/Sales?$apply=groupby((rolluprecursive(

$root/SalesOrganizations,SalesOrgHierarchy,SalesOrganization
/ID,
    ancestors($root/SalesOrganizations,SalesOrgHierarchy,ID,
              filter(ID eq 'US East'),keep start))),
  aggregate(Amount with sum as Total))
```

results in

```
{
  "@context": "$metadata#Sales(SalesOrganization(),Total)",
  "value": [
    { "SalesOrganization": { "ID": "US East", "Name": "US
East" },
      "Total@type": "Decimal", "Total": 12 },
    { "SalesOrganization": { "ID": "US",      "Name": "US"
},
      "Total@type": "Decimal", "Total": 19 },
    { "SalesOrganization": { "ID": "Sales",   "Name":
"Sales" },
      "Total@type": "Decimal", "Total": 24 }
  ]
}
```

Visual totals are computed when the ancestors transformation is additionally carried out before the rolluprecursive:

```
GET /service/Sales?$apply=

ancestors($root/SalesOrganizations,SalesOrgHierarchy,SalesOr
ganization/ID,
    filter(SalesOrganization/ID eq 'US East'),keep start))),
  /groupby((rolluprecursive(

$root/SalesOrganizations,SalesOrgHierarchy,SalesOrganization
/ID,
    ancestors($root/SalesOrganizations,SalesOrgHierarchy,ID,
              filter(ID eq 'US East'),keep start))),
  aggregate(Amount with sum as Total))
```

results in

```
{
  "@context": "$metadata#Sales(SalesOrganization(),Total)",
  "value": [
    { "SalesOrganization": { "ID": "US East", "Name": "US
East" },
      "Total@type": "Decimal", "Total": 12 },
    { "SalesOrganization": { "ID": "US",      "Name": "US"
},
      "Total@type": "Decimal", "Total": 12 },
    { "SalesOrganization": { "ID": "Sales",   "Name":
"Sales" },
      "Total@type": "Decimal", "Total": 12 }
  ]
}
```

⚠ *Example 68: Although \(p={\tt ID}\) and \(q={\tt ID}\), they are not equal in the sense of case 1, because they are evaluated relative to different entity sets. Hence, this is an example of case 3 of the definition* ~~{~~

~~"@odata.id": "aggregated-stuff2143248437259843-1",~~
~~"#Model.SingleAction": {~~
~~"title": "Do something on this entity",~~
~~"target":~~
~~"http://.../aggregated-stuff2143248437259843-1/Model.SingleAction"~~
~~},~~
~~...~~
~~},~~
~~...~~
~~}~~
~~}~~

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

~~7 Examples~~

of \(σ(x)\), where no Sales/ID matches a SalesOrganizations/ID, that is, all \(F(x)\) have empty output sets.

```
GET /service/Sales?$apply=
    groupby((rolluprecursive(

$root/SalesOrganizations,SalesOrgHierarchy,ID))),
             aggregate(Amount with sum as TotalAmount))
```

results in

```
{
  "@context":
"$metadata#Sales(SalesOrganization(),TotalAmount)",
  "value": [
    { "SalesOrganization": { "ID": "Sales", "Name":
"Corporate Sales" },
      "TotalAmount": null },
    { "SalesOrganization": { "ID": "EMEA",   "Name": "EMEA"
},
      "TotalAmount": null },
    { "SalesOrganization": { "ID": "US",     "Name": "US" },
      "TotalAmount": null },
    ...
  ]
}
```

7 Examples

The following examples show some common aggregation-related questions that can be answered by combining the transformations defined in sections 3~~chapter~~ and 6.

7.1 Requesting Distinct Values

~~7.1 Distinct Values~~

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

Example ~~:~~69:

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

results in

```
{
```

~~*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 ~~:~~70: customers that bought something

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

results in

```
{
```

~~*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*~~ Since groupby expands ~~*the*~~ navigation properties ~~*and selects the data*~~in grouping properties ~~*specified in* `groupby` *and* `aggregate`*.*~~

~~GET ~/Sales?$~~by default, this is the same result as if the request would include a $expand=Customer($select=Name~~) ~~). The groupby removes all other properties.

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 ~~:~~71:

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

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

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

This could also have been formulated as

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

Example 72: *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.~~

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

results in

```
{
  "@context": "$metadata#Sales(Customer())",
  "value": [
```

```
    { "Customer": { "ID": "C1", "Name": "Joe", "Country":
"USA" } },
    { "Customer": { "ID": "C2", "Name": "Sue", "Country":
"USA" } },
    { "Customer": { "ID": "C3", "Name": "Sue", "Country":
"Netherlands" } }
  ]
}
```

Example ~~:~~73: the first question in the motivating example in [section 2.3](#)~~section~~ , which customers bought which products, can now be expressed as

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

and results in

```
{
  "@~~odata.~~context":
"$metadata#Sales(Customer(Name,ID),Product(Name))",
  "value": [
```

```
    {~~  "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,~~      "Product": { "Name": "Coffee"} },
    { "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": "Joe", "ID": "C1" },
      "Product": { "Name": "Sugar" } },
    { "Customer": { "Name": "Sue", "ID": "C~~3~~2" },
      "Product": { "Name": "Coffee"} },
    { "Customer": { "Name": "Sue", "ID": "C2" },
      "Product": { "Name": "Paper" } },
    { ~~"@odata.id": null,~~ "Customer": { "Name": "Sue", "ID": "C3"
},
      "Product": { "Name": "Paper" } },
    { "Customer": { "Name": "Sue", "ID": "C3" },
      "Product": { "Name": "Sugar" } }
  ]
}
```

⚠ *Example 74: grouping by properties of subtypes*

```
GET
/service/Products?$apply=groupby((SalesModel.FoodProduct/Rat
ing,

SalesModel.NonFoodProduct/RatingClass))
```

results in

```
{
```

~~7.2 Aggregation Methods~~

```
  "@context":
"$metadata#Products(SalesModel.FoodProduct/Rating,

SalesModel.NonFoodProduct/RatingClass)",
  "value": [
    { "@type": "#SalesModel.FoodProduct", "Rating": 5 },
    { "@type": "#SalesModel.FoodProduct", "Rating": null },
    { "@type": "#SalesModel.NonFoodProduct", "RatingClass":
"average" },
    { "@type": "#SalesModel.NonFoodProduct", "RatingClass":
null }
  ]
}
```

⚠ Example 75: grouping by a property of a subtype

```
GET
/service/Products?$apply=groupby((SalesModel.FoodProduct/Rat
ing))
```

results in a third group representing entities with no SalesModel.FoodProduct/Rating, including the SalesModel.NonFoodProducts:

```
{
  "@context": "$metadata#Products(@Core.AnyStructure)",
  "value": [
    { "@type": "#SalesModel.FoodProduct", "Rating": 5 },
    { "@type": "#SalesModel.FoodProduct", "Rating": null },
    { }
  ]
}
```

7.2 Standard 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 expression. Expressions defining an aggregate method ~~MUST~~ specify an alias. The aggregated values are returned in a dynamic property whose name is determined by the alias.

Example 76~~.~~:

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

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

```
{
  "@context": "$metadata#Products(Name,~~Sales(~~Total~~))",~~)",
~~  "value": [~~
~~  {"@odata.id": null,~~  "value": [
    { "Name": "Coffee", "~~Sales": [{~~Total@type": "Decimal",
"Total":   12 ~~}]},~~
~~  {"@odata.id": null,~~},
    { "Name": "Paper",   "~~Sales": [{~~Total@type": "Decimal",
"Total":    8 ~~}]},~~
~~  {"@odata.id": null,~~},
    { "Name": "Pencil",                          ~~"Sales": [{~~
"Total": null ~~}]},~~
~~  {"@odata.id": null,~~},
    { "Name": "Sugar",   "~~Sales": [{~~Total@type": "Decimal",
"Total":    4 }
~~}]}~~
  ]
}
```

Note that the base set of the request is Products, so there is a result item for product Pencil even though there are no sales ~~item. As aggregate returns exactly one result item even if there are no items to be aggregated, the Sales navigation property's value is an array with one element representing the sum over no input values, which~~items. The input set for the aggregation in the third row is \(I\) consisting of the pencil, \(p=q/r={\tt Sales}/{\tt Amount}\), \(E=\Gamma(I,q)\) is empty and \(A=\Gamma(E,r)\) is also empty. The sum over the empty collection is null.

Example 77~~*Example : 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*~~

: Alternatively, the request could ask for the aggregated amount to be nested inside a clone of Sales

```
GET ~~~/~~/service/Products?$apply=~~groupby((Name,Sales/Currency/Code),~~
~~              ~~addnested(Sales,
    aggregate(~~Sales/~~Amount with sum as Total~~))~~) as
AggregatedSales)
```

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

```
{
  "@context":
"$metadata#Products(~~Name,Sales(Total,Currency(Code)))",~~AggregatedSales())",
  "value": [
~~{ "@odata.id": null,~~
    { "ID": "P2", "Name": "Coffee", "Color": "Brown",
"TaxRate": 0.06,
      "AggregatedSales@context": "#Sales(Total)",
      "AggregatedSales": [ { "Total@type": "Decimal",
"Total": 12~~, "Currency": { "Code": "USD" } } ] },~~
~~{ "@odata.id": null, "Name:~~ } ] },
    { "ID": "P3", "Name": "Paper",  "Color": "White",
"TaxRate": 0.14,
      "AggregatedSales@context": "#Sales~~": [ { "~~(Total~~": 3,~~
~~"Currency": { "Code": "EUR" } },~~
~~        ~~)",
      "AggregatedSales": [ { "Total~~": 5, "Currency": { "Code": "USD" }~~
~~} ] },~~
~~{ "@odata.id": null, "Name:~~@type": "Decimal", "Total":  8 } ] },
    { "ID": "P4", "Name": "Pencil", "Color": "Black",
"TaxRate": 0.14,
      "AggregatedSales@context": "#Sales~~": [] },~~
~~{ "@odata.id":~~ (Total)",
      "AggregatedSales": [ {
"Total": null~~,~~ } ] },
    { "ID": "P1", "Name~~:~~": "Sugar",
~~  "~~"Color": "White", "TaxRate": 0.06,
      "AggregatedSales@context": "#Sales~~": [ { "Total":  2, "Currency":~~
~~{ "Code": "EUR" } },~~
~~          { "Total":  2, "Currency": { "Code": "USD" } } ] }~~
~~]~~(Total)",
```

```
~~{~~
```

~~Note that navigation properties 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 of the navigation properties.~~

~~*Example :*~~

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

```
~~returns the different ~~      "AggregatedSales": [ { "Total@type":
"Decimal", "Total":  4 } ] }
  ]
}
```

Example 78: To compute the aggregate as a property without nesting, use the aggregate function in `$compute` rather than the aggregate transformation in `$apply`:

```
GET /service/Products?$compute=Sales/aggregate(Amount with
sum) as Total
```

results in

```
{
```

"@~~*products sold per country:*~~

```
~~{~~
```

```
~~"@odata.~~context":
"$metadata#~~Customers(Country,Sales(Product(Name)))",~~Products(*,Total
)",
```

```
  "value": [~~"value": [~~
~~{ "@odata.id": null, "Country": "Netherlands",~~
~~"Sales": [ { "Product": { "Name": "Paper"  },~~
~~{ "Product": { "Name": "Sugar"  } ] },~~
~~{ "@odata.id": null, "Country": "USA",~~
```

```
~~"Sales": [ { "Product": {~~
    { "ID": "P2", "Name": "Coffee", "Color": "Brown",
"TaxRate": 0.06,
      "Total@type": "Decimal", "Total": 12 },
    { "ID": "P3", "Name": "Paper",  "Color": "White",
"TaxRate": 0.14,
      "Total@type": "Decimal", "Total":  8 },
    { "ID": "P4", "Name": "Pencil", "Color": "Black",
"TaxRate": 0.14,
                                      "Total": null },
    { "ID": "P1", "Name": "Sugar",  "Color": "White",
"TaxRate": 0.06,
      "Total@type": "Decimal", "Total":  4 }
  ]
}
```

The expression $it/Sales refers to the sales of the current product. Without $it, all sales of all products would be aggregated, because the input collection for the aggregate function consists of all products.

Example 79: Alternatively, join could be applied to yield a flat structure:

```
GET /service/Products?$apply=
    join(Sales as TotalSales,aggregate(Amount with sum as
Total))
    /groupby((Name,TotalSales/Total))
```

results in

```
{
  "@context": "$metadata#Products(Name,TotalSales())",
  "value": ["~~},~~
~~{ "Product":~~
```

```
    { "Name": "Coffee",
      "TotalSales@context": "#Sales(Total)/$entity",
      "TotalSales": { "Total@type": "Decimal", "Total": 12 }
},
    { "Name": "Paper~~" },~~
~~        { "Product":~~",
      "TotalSales@context": "#Sales(Total)/$entity",
      "TotalSales": { "Total@type": "Decimal", "Total":  8 }
},
    { "Name": "Sugar~~" }}}~~",
```

```
~~  }~~
~~}~~
```

```
      "TotalSales@context": "#Sales(Total)/$entity",
      "TotalSales": { "Total@type": "Decimal", "Total":  4 }
}
  ]
}
```

Applying outerjoin instead would return an additional entity for product with ID "Pencil" and TotalSales having a null value.

Example ~~:~~80:

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

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

```
{
  "@context":
"$metadata#Sales(Customer(Country),AverageAmount)",
  "value": [
    {~~  "value": [~~
~~  { "@odata.id": null,~~ "Customer": { "Country": "Netherlands" },
      "AverageAmount": 1.~~6666667 },~~
~~  { "@odata.id": null,~~6666666666666667 },
    { "Customer": { "Country": "USA" },
      "AverageAmount": 3.8 }
~~]~~
```

```
~~}~~
```

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

~~*Example :*~~

```
  ]
}
```

Here the AverageAmount is of type Edm.Double.

Example 81: $count after navigation property

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

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

```
{
  "@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 aggregated property in a related entity: the aggregated property has the same navigation path as the original value.~~

~~*Example : 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,~~@type": "Decimal",
"SalesCount": 2 },
~~     "Sales": [ { "TotalAmount": 12 } ] },~~
~~  { "@odata.id": null,~~{ "Name": "Paper",  "SalesCount~~":~~@type":
"Decimal", "SalesCount": 4~~,~~
~~    "Sales": [ { "TotalAmount":  8 } ] },~~
~~  { "@odata.id": null,~~ },
    { "Name": "Pencil", "SalesCount@type": "Decimal",
"SalesCount": 0~~,~~
~~    "Sales": [ { "TotalAmount": null } ] },~~
~~  { "@odata.id": null,~~ },
```

```
    { "Name": "Sugar",  "SalesCount@type": "Decimal",
"SalesCount": 2~~,~~ }
```

```
~~          "Sales": [ { "TotalAmount":  4 } ] }~~
~~    }~~
~~}~~
```

```
  ]
}
```

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

⚠ Example ~~:~~82: The effect of the groupby is to create transient entities and avoid in the result structural properties ~~for Sales/$count and Sales/Amount are placed differently~~other than Name.

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

results in

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

```
      "AggregatedSales": [ { "SalesCount": 0, "TotalAmount":
null } ] },
~~  { "@odata.id": null, "Name": "Sugar",~~
    ~~ ~~"{ "Name": "Sugar",
      "AggregatedSales@context": "#Sales~~": [{~~
~~"~~(SalesCount,TotalAmount)",
      "AggregatedSales": [ { "SalesCount": 2,
          "TotalAmount@type": "Decimal",  "TotalAmount":  4
} ] }
~~]~~
```

```
~~}~~
```

~~7.3 Custom Aggregates~~

~~Custom aggregates are defined through the annotation. They can be associated with either an entity type or an entity container.~~
~~A can be used by specifying the name of the custom aggregate in the `aggregate` clause.~~

~~*Example :*~~

```
  ]
}
```

The aggregate function can not only be used in $compute but also in $filter and $orderby:

Example 83: Products with an aggregated sales volume of ten or more

```
GET
~~~/~~/service/Products?$filter=Sales~~?$apply=groupby((Customer/Country),~~
~~          ~~/aggregate(Amount with sum~~ as Actual,Forecast))~~) ge 10
```

~~*results in*~~

```
~~{~~
```

~~ "@odata.~~*results in*

```
{
  "@context":
"$metadata#~~Sales(Customer(Country),Actual,Forecast))",~~Products",
  "value": [
~~  { "@odata.id": null, "Customer": {~~
    { "ID": "P2", "Name": "Coffee", "Color": "Brown",
"TaxRate": 0.06 },
    { "ID": "P3", "Name": "Paper",  "Color": "White",
"TaxRate": 0.14 }
  ]
}
```

Example 84: Customers in descending order of their aggregated sales volume

```
GET /service/Customers?$orderby=Sales/aggregate(Amount with
sum) desc
```

results in

```
{
  "@context": "$metadata#Customers",
  "value": [
    { "ID": "C2", "Name": "Sue", "Country": "~~Netherlands~~USA" },
~~  "Actual":  5, "Forecast": 4 },~~
~~  { "@odata.id": null, "Customer": {~~    { "ID": "C1", "Name": "Joe",
"Country": "USA" },
```

```
~~       "Actual": 19, "Forecast": 21 }~~
~~  ]~~
~~}~~
```

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

~~*Example :*~~

```
    { "ID": "C3", "Name": "Sue", "Country": "Netherlands" },
    { "ID": "C4", "Name": "Luc", "Country": "France" }
  ]
}
```

Example 85: Contribution of each sales to grand total sales amount

```
GET ~~~/~~/service/Sales?$compute=Amount divby
$these/aggregate(Amount with sum)
                            as Contribution
```

results in

```
{
  "@context": "$metadata#Sales(*,Contribution)",
  "value": [
    { "ID": 1, "Amount": 1, "Contribution@type": "Decimal",
                            "Contribution":
0.0416666666666667 },
    { "ID": 2, "Amount": 2, "Contribution@type": "Decimal",
                            "Contribution":
0.0833333333333333 },
    { "ID": 3, "Amount": 4, "Contribution@type": "Decimal",
                            "Contribution":
0.1666666666666667 },
    { "ID": 4, "Amount": 8, "Contribution@type": "Decimal",
                            "Contribution":
0.3333333333333333 },
    { "ID": 5, "Amount": 4, "Contribution@type": "Decimal",
                            "Contribution":
0.1666666666666667 },
    { "ID": 6, "Amount": 2, "Contribution@type": "Decimal",
                            "Contribution":
0.0833333333333333 },
    { "ID": 7, "Amount": 1, "Contribution@type": "Decimal",
```

```
                            "Contribution":
0.0416666666666667 },
    { "ID": 8, "Amount": 2, "Contribution@type": "Decimal",
                            "Contribution":
0.0833333333333333 }
  ]
}
```

Example 86: Product categories with at least one product having an aggregated sales amount greater than 10

```
GET /service/Categories?$filter=Products/any(
                                p:p/Sales/aggregate(Amount
with sum) gt 10)
```

results in

```
{
  "@context": "$metadata#Categories",
  "value": [
    { "ID": "PG1", "Name": "Food" }
  ]
}
```

The aggregate function can also be applied inside $apply:

Example 87: Sales volume per customer in relation to total volume

```
GET /service/Sales?$apply=
    groupby((Customer),aggregate(Amount with sum as
CustomerAmount))
    /compute(CustomerAmount divby
$these/aggregate(CustomerAmount with sum)
             as Contribution)
  &$expand=Customer/$ref
```

results in

```
{
  "@context":
"$metadata#Sales(Customer(),CustomerAmount,Contribution)",
  "value": [
    { "Customer":    { "@id": "Customers('C1')" },
      "Contribution@type": "Decimal", "Contribution":
0.2916667 },
    { "Customer":    { "@id": "Customers('C2')" },
      "Contribution@type": "Decimal", "Contribution": 0.5 },
    { "Customer":    { "@id": "Customers('C3')" },
      "Contribution@type": "Decimal", "Contribution":
0.2083333 }
  ]
}
```

Example 88: rule 1 for keyword from applied repeatedly

```
GET /service/Sales?$apply=aggregate(Amount with sum
                                    from Time with average
                                    from Customer/Country
with max
                                    as
MaxDailyAveragePerCountry)
```

is equivalent to (with nested groupby transformations)

```
GET /service/Sales?$apply=
  groupby((~~Name),~~
              Customer/Country),
    groupby((Time),aggregate(Amount with sum as D1))
    /aggregate(D1 with average as D2))
  /aggregate(D2 with max as MaxDailyAveragePerCountry)
```

and is equivalent to (with consecutive groupby transformations)

```
GET /service/Sales?$apply=
  groupby((Customer/Country,Time),aggregate(Amount with sum
as D1))
  /groupby((Customer/Country),aggregate(D1 with average as
D2))
  /aggregate(D2 with max as MaxDailyAveragePerCountry)
```

7.3 Requesting Expanded Results

Example 89: Assuming an extension of the data model where Customer contains an additional collection-valued complex property Addresses and these contain a single-valued navigation property ResponsibleSalesOrganization, addnested can be used to compute a nested dynamic property:

```
GET /service/Customers?$apply=
    addnested(Addresses/ResponsibleSalesOrganization,
              compute(Superordinate/Name as SalesRegion)
              as AugmentedSalesOrganization)
```

results in

```
{
  "@context":
"$metadata#Customers(Addresses(AugmentedSalesOrganization())
",
  "value": [
    { "ID": "C1", "Name": "Joe", "Country": "US",
      "Addresses": [
        { "Locality": "Seattle",
          "AugmentedSalesOrganization":
          { "@context": "#SalesOrganizations/$entity",
            "ID": "US West", "SalesRegion": "US" } },
        { "Locality": "DC",
          "AugmentedSalesOrganization":
          { "@context": "#SalesOrganizations/$entity",
            "ID": "US",      "SalesRegion": "Corporate
Sales" } },
      ]
    }, ...
  ]
}
```

addnested transformations can be nested.

Example 90: nested addnested transformations

```
GET /service/Categories?$apply=
    addnested(Products,
      addnested(Sales,filter(Amount gt 3) as FilteredSales)
    as FilteredProducts)
```

results in

```
{
  "@context": "$metadata#Categories(FilteredProducts()",
  "value": [
    { "ID": "PG1", "Name": "Food",
      "FilteredProducts@context":
"#Products(FilteredSales())",
      "FilteredProducts": [
        { "ID": "P1", "Name": "Sugar",  "Color": "White",
          "FilteredSales@context": "#Sales",
          "FilteredSales": [] },
        { "ID": "P2", "Name": "Coffee", "Color": "Brown",
          "FilteredSales@context": "#Sales~~(Amount with sum as
Actual),~~
                  ",
          "FilteredSales": [ { "ID": 3, "Amount": 4 },
                             { "ID": 4, "Amount": 8 } ] }
      ]
    },
    { "ID": "PG2", "Name": "Non-Food",
      "FilteredProducts@context":
"#Products(FilteredSales())",
      "FilteredProducts": [
        { "ID": "P3", "Name": "Paper",  "Color": "White",
          "FilteredSales@context": "#Sales~~/Forecast))~~",
          "FilteredSales": [ { "ID": 5, "Amount": 4 } ] },
        { "ID": "P4", "Name": "Pencil", "Color": "Black",
          "FilteredSales@context": "#Sales",
          "FilteredSales": [] }
      ]
    }
  ]
}
```

Instead of keeping all related entities from navigation properties that addnested expanded by default, an explicit $expand controls which of them to include in the response:

```
GET /service/Categories?$apply=
    addnested(Products,
      addnested(Sales,filter(Amount gt 3) as FilteredSales)
    as FilteredProducts)
  &$expand=FilteredProducts
```

results in the response before without the FilteredSales dynamic navigation properties expanded in the result.

Example 91: Here only the GroupedSales are expanded, because they are named in $expand, the related Product entity is not:

```
GET /service/Customers?$apply=addnested(Sales,
    groupby((Product/Name)) as GroupedSales)
  &$expand=GroupedSales
```

results in

```
{
  "@context": "$metadata#Customers(GroupedSales())",
  "value": [
    { "ID": "C1", "Name": "Joe", "Country": "USA",
      "GroupedSales@context": "#Sales(@Core.AnyStructure)",
      "GroupedSales": [
        { },
        { },
        { }
      ] },
    { "ID": "C2", "Name": "Sue", "Country": "USA",
      "GroupedSales@context": "#Sales(@Core.AnyStructure)",
      "GroupedSales": [
        { },
        { }
      ] },
    { "ID": "C3", "Name": "Joe", "Country": "Netherlands",
      "GroupedSales@context": "#Sales(@Core.AnyStructure)",
      "GroupedSales": [
        { },
        { }
      ] },
    { "ID": "C4", "Name": "Luc", "Country": "France",
      "GroupedSales@context": "#Sales(@Core.AnyStructure)",
      "GroupedSales": [ ] }
  ]
}
```

Example 92: use outerjoin to split up collection-valued navigation properties for grouping

```
GET /service/Customers?$apply=outerjoin(Sales as
ProductSales)

/groupby((Country,ProductSales/Product/Name))
```

returns the different combinations of products sold per country:

```
{
  "@context":"$metadata#Customers(Country,ProductSales())",
  "value": [
    { "Country": "Netherlands",
      "ProductSales@context":
"#Sales(Product(Name))/$entity",
      "ProductSales": { "Product": { "Name": "Paper"  } } },
    { "Country": "Netherlands",
      "ProductSales@context":
"#Sales(Product(Name))/$entity",
      "ProductSales": { "Product": { "Name": "Sugar"  } } },
    { "Country": "USA",
      "ProductSales@context":
"#Sales(Product(Name))/$entity",
      "ProductSales": { "Product": { "Name": "Coffee" } } },
    { "Country": "USA",
      "ProductSales@context":
"#Sales(Product(Name))/$entity",
      "ProductSales": { "Product": { "Name": "Paper"  } } },
    { "Country": "USA",
      "ProductSales@context":
"#Sales(Product(Name))/$entity",
      "ProductSales": { "Product": { "Name": "Sugar"  } } },
    { "Country": "France", "ProductSales": null }
  ]
}
```

7.4 Requesting Custom Aggregates

Custom aggregates are defined through the CustomAggregate annotation. They can be associated with an entity set, a collection or an entity container.

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

Example 93:

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

results in

```
{
  "@~~odata.~~context":
"$metadata#~~Products(Name,~~Sales(Customer(Country),Actual,Foreca
st~~))",~~)",
  "value": [
~~{"@odata.id":null,"Name":"Coffee","Sales":[{"~~
    { "Customer": { "Country": "Netherlands" },
      "Actual~~":12,"Forecast":6}] },~~
~~  { "@odata.id":null,"Name":"Paper", "Sales":[{"~~@type": "Decimal",
"Actual": ~~8,"~~ 5,
      "Forecast@type": "Decimal", "Forecast":~~2}] },~~
~~  { "@odata.id":null,"Name":"Pencil","Sales":[] },~~
~~  { "@odata.id":null,"Name":"Sugar", "Sales":[{"~~ 4 },
    { "Customer": { "Country": "USA" },
      "Actual~~": 4,"~~@type": "Decimal", "Actual": 19,
      "Forecast~~":7}] }~~
~~]~~@type": "Decimal", "Forecast": 21 }
  ]
}
```

When associated with an entity ~~type~~set a custom aggregate MAY have the same name as a property of the underlying entity type 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 94: 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
~~~/~~/service/Sales?$apply=groupby((Customer/Country),aggregate(
Amount))
```

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

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

Example 95: illustrates rule 1 for keyword from: maximal sales forecast for a product

```
GET /service/Sales?$apply=aggregate(Forecast from Product
with max
                                    as MaxProductForecast)
```

is equivalent to

```
GET /service/Sales?$apply=
  groupby((Product),aggregate(Forecast))
  /aggregate(Forecast with max as MaxProductForecast)
```

Example 96: illustrates rule 2 for keyword from: the forecast is computed in two steps

```
GET /service/Sales?$apply=aggregate(Forecast from Product as
ProductForecast)
```

is equivalent to the following (except that the property name is Forecast instead of ProductForecast)

```
GET /service/Sales?$apply=
  groupby((Product),aggregate(Forecast))
  /aggregate(Forecast)
```

~~7.4~~ Example 97: illustrates rule 1 followed by rule 2 for keyword from~~Aliasing~~

: a forecast based on the average daily forecasts per country

```
GET /service/Sales?$apply=aggregate(Forecast from Time with
average
                                    from Customer/Country
                                    as CountryForecast)
```

is equivalent to the following (except that the property name is Forecast instead of CountryForecast). Note that Forecast appears as a property and as a custom aggregate.

```
GET /service/Sales?$apply=
  groupby((Customer/Country),
    groupby((Time),aggregate(Forecast))
    /aggregate(Forecast with average as D1))
  /aggregate(Forecast)
```

7.5 Aliasing

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

Example ~~:~~98:

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

results in

```
{
```

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

The introduced dynamic ~~properties MUST always be in the same set as the original~~ property~~.~~ is added to the context where the aggregate expression is applied to:

Example ~~:~~99:

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

~~results in~~

```
~~{~~
```

```
~~"@odata.~~               as AggregatedSales))
```

results in

```
{
  "@context":
"$metadata#Products(Name,~~Sales(~~Total,~~AvgAmt))",~~AggregatedSales()
)",
  "value": [
~~{"@odata.id":null,"~~
    { "Name~~":"~~": "Coffee~~","Sales":[{"~~", "Total":   12~~,"~~,
      "AggregatedSales@context": "#Sales(AvgAmt~~":~~)",
      "AggregatedSales": [ { "AvgAmt@type": "Decimal",
                             "AvgAmt": 6~~}]},~~
~~{"@odata.id":null,"~~ } ] },
    { "Name~~":"~~": "Paper", ~~"Sales":[{"~~ "Total":    8~~,"~~,
      "AggregatedSales@context": "#Sales(AvgAmt~~":~~)",
      "AggregatedSales": [ { "AvgAmt@type": "Decimal",
                             "AvgAmt": 2~~}]},~~
~~{"@odata.id":null,"~~ } ] },
```

```
    { "Name~~":"~~": "Pencil~~","~~", "Total": null,
        "AggregatedSales@context":
"#Sales~~":[{"Total":null,"AvgAmt":null}]},~~
~~  {"@odata.id":null,"~~(AvgAmt)",
        "AggregatedSales": [ { "AvgAmt": null } ] },
    { "Name~~":"~~": "Sugar", ~~"Sales":[{"~~ "Total":    4~~,"~~,
        "AggregatedSales@context": "#Sales(AvgAmt~~":~~)",
        "AggregatedSales": [ { "AvgAmt@type": "Decimal",
                               "AvgAmt": 2~~}]}~~
 } ] }
  ]
}
```

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

Example ~~:~~100:

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

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

```
{
  "@~~odata.~~context": "$metadata#Sales(Amount,Total)",
  "value": [
~~  {"@odata.id": null,~~
    { "Amount": 1, "Total@type": "Decimal", "Total": 2 },
~~  {"@odata.id": null,~~
    { "Amount": 2, "Total@type": "Decimal", "Total": 6 },
~~  {"@odata.id": null,~~
    { "Amount": 4, "Total@type": "Decimal", "Total": 8 },
~~  {"@odata.id": null,~~
    { "Amount": 8, "Total@type": "Decimal", "Total": 8 }
~~}~~
  ]
}
```

7.6 Combining Transformations per Group~~}~~

~~7.5 Combining Transformations per Group~~

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

Example 101: 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 ~~~/~~/service/Sales?$apply=concat(

groupby((Customer/Country,Product/Name~~,Currency/Code),~~
),
                              aggregate(Amount with sum as
Total))
                /groupby((Customer/Country~~,Currency/Code),~~
               ),topcount(1,Total)),
                      groupby((Customer/Country~~,Currency/Code~~),
                              aggregate(Amount with sum as
Total)))
```

results in

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

Example ~~:~~102: transformation sequences are also useful inside groupby*: ~~To get~~Aggregate the~~ aggregated~~ amount by only considering the top two sales amounts per product and country:*

```
GET
~~~/~~/service/Sales?$apply=groupby((Customer/Country,Product/Nam
e~~,Currency/Code~~),
                      topcount(2,Amount)/aggregate(Amount
with sum as Total))
```

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

```
{
  "@context":
"$metadata#Sales(Customer(Country),Product(Name),Total~~,Currenc
y(Code))",~~)",
  "value": [
~~  { "@odata.id": null,~~
    { "Customer~~": {~~":{ "Country": "Netherlands" }, "Product~~":
{~~":{ "Name": "Paper" },
      "Total@type": "Decimal", "Total":  3~~, "Currency": { "Code":
"EUR" }~~
    },
~~  { "@odata.id": null,~~
    { "Customer~~": {~~":{ "Country": "Netherlands" }, "Product":{
"Name": "Sugar" },
      "Total@type": "Decimal", "Total":  2
    },
    { "Customer":{ "Country": "USA" },        "Product~~": {~~":{
"Name": "Sugar" },
      "Total@type": "Decimal", "Total":  2~~, "Currency": { "Code":
"EUR" }~~
    },
~~  { "@odata.id": null,~~
    { "Customer~~": {~~":{ "Country": "USA" },          "Product~~":
{~~":{ "Name": "Coffee" },
      "Total@type": "Decimal", "Total": 12~~, "Currency": { "Code":
"USD" }~~
    },
~~  { "@odata.id": null,~~
    { "Customer~~": {~~":{ "Country": "USA" },          "Product~~":
{~~":{ "Name": "Paper" },
      "Total@type": "Decimal", "Total":  5~~, "Currency": { "Code":
"USD" }~~
    }
~~]~~
  ]
}
```

Example 103~~}~~

~~7.6 Model Functions as Set Transformations~~

: concatenation of two different groupings "biggest sale per customer" and "biggest sale per product", made distinguishable by a dynamic property:

```
GET /service/Sales?$apply=concat(
    groupby((Customer),topcount(1,Amount))/compute('Customer' as per),
    groupby((Product),topcount(1,Amount))/compute('Product' as per))
  &$expand=Customer($select=ID),Product($select=ID)
```

In the result, Sales entities 4 and 6 occur twice each with contradictory values of the dynamic property per. If a UI consuming the response presents the two groupings in separate columns based on the per property, no contradiction effectively arises.

```
{
  "@context": "$metadata#Sales(*,per,Customer(ID),Product(ID))",
  "value": [
    { "Customer": { "ID": "C1" }, "Product": { "ID": "P2" },
      "ID": "3", "Amount": 4, "per": "Customer" },
    { "Customer": { "ID": "C2" }, "Product": { "ID": "P2" },
      "ID": "4", "Amount": 8, "per": "Customer" },
    { "Customer": { "ID": "C3" }, "Product": { "ID": "P1" },
      "ID": "6", "Amount": 2, "per": "Customer" },
    { "Customer": { "ID": "C3" }, "Product": { "ID": "P1" },
      "ID": "6", "Amount": 2, "per": "Product" },
    { "Customer": { "ID": "C2" }, "Product": { "ID": "P2" },
      "ID": "4", "Amount": 8, "per": "Product" },
    { "Customer": { "ID": "C2" }, "Product": { "ID": "P3" },
      "ID": "5", "Amount": 4, "per": "Product" }
  ]
}
```

7.7 Model Functions as Set Transformations

Example ~~: as~~104: As a variation of example 101~~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 TopCountAnd~~B~~Rema~~la~~in~~c~~der function that accepts ~~the~~a count and a numeric property for the top entities~~ in the given input set not to be considered for the balance~~:

```
<edm:Function Name="TopCountAnd~~B~~Rema~~la~~in~~c~~der"
              ~~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:ReturnType Type="Collection(Edm.EntityType)" />
</edm:Function>
```

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

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

Self.TopCountAnd~~B~~Rema~~la~~in~~c~~der(Count=1,Property='Total'))
```

results in

```
{
  "@~~odata.~~context":
"$metadata#Sales(Customer(Country),~~Product(Name),~~Total)",
  "value": [
~~  { "@odata.id": null,~~
    { "Customer": { "Country": "Netherlands" },
      "Product": { "Name": "Paper" },
      "Total@type": "Decimal", "Total":  3 },
~~  { "@odata.id": null,~~
    { "Customer": { "Country": "Netherlands" },
      "~~Product": { "Name": "**Other**" },~~Total@type": "Decimal",
"Total":  2 },
~~  { "@odata.id": null,~~
    { "Customer": { "Country": "USA" },
      "Product": { "Name": "Coffee" },
      "Total@type": "Decimal", "Total": 12 },
~~  { "@odata.id": null,~~
    { "Customer": { "Country": "USA" },
      "~~Product": { "Name": "**Other**" }, "~~Total~~": 5 }~~
@type": "Decimal", "Total":  7 }
  ]
}
```

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

~~7.7~~ 7.8 Controlling Aggregation per Rollup Level~~Controlling Aggregation per Rollup Level~~

~~Consumers~~

For a leveled hierarchy, consumers may specify a different aggregation method per level for every property passed to rollup as a hierarchy level below the root level.

Example ~~:~~105: get the average of the overall amount by month per product.

Using a transformation sequence:

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

Using ~~:~~from:

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

Example ~~: for an aggregate entity set listing~~106: get the total ~~sales amounts~~ amount per customer~~ and country~~, the ~~rollup shall produce additional instances for the~~ average of the total ~~sales amount of customers~~customer amounts per country, and the overall average of ~~that average (which is a bit boring because the example data doesn't have two countries with the same currency☺)~~these averages

```
GET ~~~/~~/service/Sales?$apply=concat(

groupby((rollup~~($all,~~(Customer/Country,Customer/ID~~),~~
             ~~Currency/Code),~~
)),
                           aggregate(Amount with sum
~~ as CustomerCountryAverage~~
                                    from Customer/ID with
average
                                    as
CustomerCountryAverage)),
                    aggregate(Amount with sum
                              from Customer/ID      with
average
```

```
                              from Customer/Country with
average~~))~~
```

~~results in~~

```
~~{~~
```

```
~~ "@odata.~~                              as
CustomerCountryAverage)))
```

results in

```
{
   "@context":
"$metadata#Sales(~~Customer(Country,ID),~~CustomerCountryAverage~~,Curre
ncy(Code))",~~)",
   "value": [
~~   { "@odata.id": null,~~
      { "Customer": { "Country": "USA", "ID": "C1" },
        "CustomerCountryAverage@type":"Decimal",
        "CustomerCountryAverage":    7~~, "Currency": { "Code": "USD" }~~
~~   },~~
~~   { "@odata.id": null,~~ },
      { "Customer": { "Country": "USA", "ID": "C2" },
        "CustomerCountryAverage@type":"Decimal",
        "CustomerCountryAverage":   12~~, "Currency": { "Code": "USD" }~~
~~   },~~
~~   { "@odata.id": null,~~ },
      { "Customer": { "Country": "USA" },
        "CustomerCountryAverage@type":"Decimal",
        "CustomerCountryAverage": 9.5~~, "Currency": { "Code": "USD" }~~
~~   },~~
~~   { "@odata.id": null,~~ },
      { "Customer": { "Country": "Netherlands", "ID": "C3" },
        "CustomerCountryAverage~~": ~~@type":"Decimal",
        "CustomerCountryAverage": 5~~, "Currency": { "Code": "EUR" }~~
~~   },~~
~~   { "@odata.id": null,~~ },
      { "Customer": { "Country": "Netherlands" },
        "CustomerCountryAverage~~": ~~@type":"Decimal",
        "CustomerCountryAverage": 5 },
      { "CustomerCountryAverage@type":"Decimal",
        "CustomerCountryAverage": 7.25 }
   ]
}
```

Note that this example extends the result of `rollup` with `concat` and `aggregate` to append the overall average.

7.9 Aggregation in Recursive Hierarchies

If aggregation along a recursive hierarchy does not apply to the entire hierarchy, transformations ancestors and descendants may be used to restrict it as needed.

Example 107: Total sales amounts for sales orgs in 'US' in the SalesOrgHierarchy defined in Hierarchy Examples

```
GET /service/Sales?$apply=
    descendants(

$root/SalesOrganizations,SalesOrgHierarchy,SalesOrganization
/ID,
        filter(SalesOrganization/Name eq 'US'),keep start)
    /groupby((rolluprecursive(

$root/SalesOrganizations,SalesOrgHierarchy,SalesOrganization
/ID)),
      aggregate(Amount with sum as TotalAmount))
  &$expand=SalesOrganization($expand=Superordinate/$ref)
```

results in

```
{
  "@context":
"$metadata#Sales(TotalAmount,SalesOrganization())",
  "value": [
    { "TotalAmount@type": "Decimal", "TotalAmount": 19,
      "SalesOrganization": { "ID": "US",       "Name": "US",
        "Superordinate": { "@id":
"SalesOrganizations('Sales')" } } },
    { "TotalAmount@type": "Decimal", "TotalAmount": 12,
      "SalesOrganization": { "ID": "US East", "Name": "US
East",
        "Superordinate": { "@id": "SalesOrganizations('US')"
} } },
    { "TotalAmount@type": "Decimal", "TotalAmount":  7,
      "SalesOrganization": { "ID": "US West", "Name": "US
West",
        "Superordinate": { "@id": "SalesOrganizations('US')"
} } }
  ]
}
```

Note that this example returns the actual total sums regardless of whether the descendants transformation comes before or after the groupby with rolluprecursive.

The order of transformations becomes relevant if groupby with rolluprecursive shall aggregate over a thinned-out hierarchy, like here:

Example 108: Number of Paper sales per sales org aggregated along the the SalesOrgHierarchy defined in Hierarchy Examples~~, "Currency": { "Code": "EUR" }~~

~~},~~
~~{"@odata.~~

```
GET /service/Sales?$apply=
    filter(Product/Name eq 'Paper')
    /groupby((rolluprecursive((

$root/SalesOrganizations,SalesOrgHierarchy,SalesOrganization
/ID)),
      aggregate($count as PaperSalesCount))
  &$expand=SalesOrganization($expand=Superordinate/$ref)
```

results in

```
{
  "@context":
"$metadata#Sales(PaperSalesCount,SalesOrganization())",
  "value": [
    { "PaperSalesCount@type": "Decimal", "PaperSalesCount":
2,
      "SalesOrganization": { "ID": "US",           "Name":
"US",
        "Superordinate": { "@id":
"SalesOrganizations('Sales')" } } },
    { "PaperSalesCount@type": "Decimal", "PaperSalesCount":
1,
      "SalesOrganization": { "ID": "US East",      "Name":
"US East",
        "Superordinate": { "@id": "SalesOrganizations('US')"
} } },
    { "PaperSalesCount@type": "Decimal", "PaperSalesCount":
1,
      "SalesOrganization": { "ID": "US West",      "Name":
"US West",
        "Superordinate": { "@id": ~~null,~~
"SalesOrganizations('US')" } } },
    { "PaperSalesCount@type": "Decimal", "PaperSalesCount":
2,
      "SalesOrganization": { "ID": "EMEA",         "Name":
"EMEA",
        "Superordinate": { "@id":
"SalesOrganizations('Sales')" } } },
    { "PaperSalesCount@type": "Decimal", "PaperSalesCount":
2,
      "SalesOrganization": { "ID": "EMEA Central", "Name":
"EMEA Central",
        "Superordinate": { "@id":
"SalesOrganizations('EMEA')" } } },
    { "PaperSalesCount@type": "Decimal", "PaperSalesCount":
4,
      "SalesOrganization": { "ID": "Sales",        "Name":
"Sales",
        "Superordinate": null } }
  ]
}
```

⚠ Example 109: The input set Sales is filtered along a hierarchy on a related entity (navigation property SalesOrganization) before an aggregation

```
GET /service/Sales?$apply=
  descendants($root/SalesOrganizations,
    SalesOrgHierarchy,
    SalesOrganization/ID,
    filter(SalesOrganization/Name eq 'US'),
    keep start)
  /aggregate(Amount with sum as TotalAmount)
```

The same aggregate value is computed if the input set is the hierarchical entity SalesOrganizations and an assumed partner navigation property Sales of SalesOrganization appears in the aggregate transformation

```
GET /service/SalesOrganizations?$apply=
  descendants($root/SalesOrganizations,
    SalesOrgHierarchy,
    ID,
    filter(Name eq 'US'),
    keep start)
  /aggregate(Sales/Amount with sum as TotalAmount)
```

⚠ Example 110: total sales amount aggregated along the sales organization sub-hierarchy with root EMEA restricted to 3 levels

```
GET /service/Sales?$apply=
  groupby((rolluprecursive($root/SalesOrganizations,
                           SalesOrgHierarchy,
                           SalesOrganization/ID)),
          aggregate(Amount with sum as Total))
  /filter(Aggregation.isdescendant(
    HierarchyNodes=$root/SalesOrganizations,
    HierarchyQualifier='SalesOrgHierarchy',
    Node=SalesOrganization/ID,
    Ancestor='EMEA',
    MaxDistance=2,
    IncludeSelf=true))
  /orderby(SalesOrganization/Name)
  /traverse($root/SalesOrganizations,
            SalesOrgHierarchy,SalesOrganization/ID,preorder)
```

or, equivalently

```
GET /service/Sales?$apply=
  groupby((rolluprecursive(
    $root/SalesOrganizations,
    SalesOrgHierarchy,
    SalesOrganization/ID,
    descendants(
      $root/SalesOrganizations,
      SalesOrgHierarchy,
      ID,
      filter(ID eq 'EMEA'),
      2, keep start))),
  aggregate(Amount with sum as Total))
  /orderby(SalesOrganization/Name)
  /traverse($root/SalesOrganizations,
            SalesOrgHierarchy,SalesOrganization/ID,preorder)
```

Example 111: Return the result of example 66 in preorder

```
GET /service/Sales?$apply=groupby(
    (rolluprecursive(
      $root/SalesOrganizations,
      SalesOrgHierarchy,
      SalesOrganization/ID,
      descendants(
        $root/SalesOrganizations,
        SalesOrgHierarchy,
        ID, filter(ID eq 'US'), keep start))),
    compute(case(SalesOrganization eq
Aggregation.rollupnode():Amount)
            as AmountExcl)
    /aggregate(Amount with sum as TotalAmountIncl,
               AmountExcl with sum as TotalAmountExcl))
    /traverse($root/SalesOrganizations,
              SalesOrgHierarchy,
              SalesOrganization/ID,
              preorder,
              Name asc)
```

results in

```
{
  "@context": "$metadata#Sales(SalesOrganization(ID),

TotalAmountIncl,TotalAmountExcl)",
  "value": [
    { "SalesOrganization": { "ID": "US",       "Name": "US"
},
      "TotalAmountIncl@type": "Decimal", "TotalAmountIncl":
19,
      "TotalAmountExcl": null },
    { "SalesOrganization": { "ID": "US East", "Name": "US
East" },
      "TotalAmountIncl@type": "Decimal", "TotalAmountIncl":
12,
      "TotalAmountExcl@type": "Decimal", "TotalAmountExcl":
12 },
    { "SalesOrganization": { "ID": "US West", "Name": "US
West" },
      "TotalAmountIncl@type": "Decimal", "TotalAmountIncl":
7,
      "TotalAmountExcl@type": "Decimal" ,"TotalAmountExcl":
7 }
  ]
}
```

Example 112: Preorder traversal of a hierarchy with 1:N relationship with collection-valued segment \(p_1={\tt Sales}\) and \(r={\tt SalesOrganization}/{\tt ID}\).

```
GET /service/Products?$apply=traverse(
      $root/SalesOrganizations,
      SalesOrgHierarchy,
      Sales/SalesOrganization/ID,
      preorder,
      Name asc)
    &$select=ID
```

The result contains multiple instances of the same Product that differ in their Sales navigation property even though they agree in their ID key property. The node \(x\) with \(x/{\tt ID}={}\)"US" has \(σ(x)={}\){"Sales": [{"SalesOrganization": {"ID": "US"}}]}.

```
{
  "@context":
      "$metadata#Products(ID,Sales(SalesOrganization(ID)))",
  "value": [
    { "ID": "P1", "Sales": [ { "SalesOrganization": { "ID":
"Sales" } } ] },
    { "ID": "P2", "Sales": [ { "SalesOrganization": { "ID":
"Sales" } } ] },
    { "ID": "P3", "Sales": [ { "SalesOrganization": { "ID":
"Sales" } } ] },
    { "ID": "P1", "Sales": [ { "SalesOrganization": { "ID":
"EMEA" } } ] },
    { "ID": "P3", "Sales": [ { "SalesOrganization": { "ID":
"EMEA" } } ] },
    { "ID": "P1",
      "Sales": [ { "SalesOrganization": { "ID": "EMEA
Central" } } ] },
    { "ID": "P3",
      "Sales": [ { "SalesOrganization": { "ID": "EMEA
Central" } } ] },
    { "ID": "P1", "Sales": [ { "SalesOrganization": { "ID":
"US" } } ] },
    { "ID": "P2", "Sales": [ { "SalesOrganization": { "ID":
"US" } } ] },
    { "ID": "P3", "Sales": [ { "SalesOrganization": { "ID":
"US" } } ] },
    { "ID": "P2", "Sales": [ { "SalesOrganization": { "ID":
"US East" } } ] },
    { "ID": "P3", "Sales": [ { "SalesOrganization": { "ID":
"US East" } } ] },
    { "ID": "P1", "Sales": [ { "SalesOrganization": { "ID":
"US West" } } ] },
    { "ID": "P2", "Sales": [ { "SalesOrganization": { "ID":
"US West" } } ] },
    { "ID": "P3", "Sales": [ { "SalesOrganization": { "ID":
"US West" } } ] }
  ]
}
```

Example 113: Aggregation along a hierarchy with 1:N relationship: Sold products per sales organization

```
GET /service/Products?$apply=
    groupby((rolluprecursive(
               $root/SalesOrganizations,
               SalesOrgHierarchy,
               Sales/SalesOrganization/ID)),
             aggregate(ID with Custom.concat as
SoldProducts)
```

results in

```
{
  "@context":
"$metadata#Products(Sales(SalesOrganization(ID)),SoldProduct
s)",
  "value": [
    { "Sales": [ { "SalesOrganization": { "ID": "Sales" } }
],
      "SoldProducts": "P1,P2,P3" },
    { "Sales": [ { "SalesOrganization": { "ID": "EMEA" } }
],
      "SoldProducts": "P1,P3" },
    { "Sales": [ { "SalesOrganization": { "ID": "EMEA
Central" } } ],
      "SoldProducts": "P1,P3" },
    { "Sales": [ { "SalesOrganization": { "ID": "US" } } ],
      "SoldProducts": "P1,P2,P3" },
    { "Sales": [ { "SalesOrganization": { "ID": "US East" }
} ],
      "SoldProducts": "P2,P3" },
    { "Sales": [ { "SalesOrganization": { "ID": "US West" }
} ],
      "SoldProducts": "P1,P2,P3" }
  ]
}
```

⚠ Example 114: Assume an extension of the data model where a SalesOrganization is associated with one or more instances of ProductCategory, and ProductCategory also organizes categories in a recursive hierarchy:

ProductCategory	parent ProductCategory	associated SalesOrganizations
Food		US, EMEA
Cereals	Food	US
Organic cereals	Cereals	US West

Aggregation of sales amounts along the sales organization hierarchy could be restricted to those organizations linked with product category "Cereals" or a descendant of it, and the ancestors of those organizations:

```
GET /service/Sales?$apply=groupby((rolluprecursive(
    $root/SalesOrganizations,SalesOrgHierarchy,
    SalesOrganization/ID,
    ancestors(
      $root/SalesOrganizations,SalesOrgHierarchy,
      ID,
      traverse(
        $root/ProductCategories,ProductCategoryHierarchy,
        ProductCategories/ID,
        preorder,
        filter(Name eq 'Cereals')),
      keep start)
    )),
    aggregate(Amount with sum as TotalAmount))

&$expand=SalesOrganization($select=ID,$expand=ProductCategor
ies/$ref)
```

results in

```
{
  "@context":
"$metadata#Sales(SalesOrganization(ID),TotalAmount)",
  "value": [
    { "SalesOrganization": { "ID": "Sales",
"ProductCategories": [ ] },
      "TotalAmount@type": "Decimal", "TotalAmount": 24 },
    { "SalesOrganization": { "ID": "US",
"ProductCategories": [
      { "@id": "ProductCategories('Food')" },
      { "@id": "ProductCategories('Cereals')" } ] },
      "TotalAmount@type": "Decimal", "TotalAmount": 19 },
    { "SalesOrganization": { "ID": "US West",
"ProductCategories": [
      { "@id": "ProductCategories('Organic cereals')" } ] },
      "TotalAmount@type": "Decimal", "TotalAmount":  7 }
  ]
}
```

traverse acts here as a filter, hence preorder could be changed to postorder without changing the result. filter is the parameter \(S\) of traverse and operates on the product category hierarchy being traversed.

Replacing the traverse transformation with a descendants transformation, as in

```
ancestors(
  $root/SalesOrganizations,SalesOrgHierarchy,
  ID,
  descendants(
    $root/ProductCategories,ProductCategoryHierarchy,
    ProductCategories/ID,
    filter(ProductCategories/any(c:c/Name eq 'Cereals')),
    keep start),
  keep start)
```

works differently: descendants is the parameter \(T\) of ancestors and operates on its input set of sales organizations. This would determine descendants of sales organizations for "Cereals" and their ancestor sales organizations, so US East would appear in the result.

7.10 Maintaining Recursive Hierarchies

Besides changes to the structural properties of the entities in a hierarchical collection, hierarchy maintenance involves changes to the parent-child relationships.

Example 115: Move a sales organization Switzerland under the parent EMEA Central by binding the parent navigation property to EMEA Central ODdata-JSON, section 8.5:

```
PATCH /service/SalesOrganizations('Switzerland')
Content-Type: application/json

{ "Superordinate": { "@id": "SalesOrganizations('EMEA Central')" } }
```

results in 204 No Content.

Deleting the parent from the sales organization Switzerland (making it a root) can be achieved either with:

```
PATCH /service/SalesOrganizations('Switzerland')
Content-Type: application/json

{ "Superordinate": { "@id": null } }
```

or with:

```
DELETE /service/SalesOrganizations('Switzerland')/Superordinate/$ref
```

Example 116: If the parent navigation property contained a referential constraint for the key of the target OData-CSDL, section 8.5,

```
<EntityType Name="SalesOrganization">
  <Key>
    <PropertyRef Name="ID" />
  </Key>
  <Property Name="ID" Type="Edm.String" Nullable="false" />
  <Property Name="Name" Type="Edm.String" />
  <Property Name="SuperordinateID" Type="Edm.String" />
  <NavigationProperty Name="Superordinate"
                      Type="SalesModel.SalesOrganization">
    <ReferentialConstraint Property="SuperordinateID"
                           ReferencedProperty="ID" />
  </NavigationProperty>
</EntityType>
```

then alternatively the property taking part in the referential constraint OData-Protocol, section 11.4.9.1 could be changed to EMEA Central:

```
PATCH /service/SalesOrganizations('Switzerland')
Content-Type: application/json

{ "SuperordinateID": "EMEA Central" }
```

If the parent-child relationship between sales organizations is maintained in a separate entity set, a node can have multiple parents, with additional information on each parent-child relationship.

⚠ Example 117: Assume the relation from a node to its parent nodes contains a weight:

```
<EntityType Name="SalesOrganizationRelation">
  <Key>
    <PropertyRef Name="Superordinate/ID"
Alias="SuperordinateID" />
  </Key>
  <Property Name="Weight" Type="Edm.Decimal"
                          Nullable="false" DefaultValue="1"
/>
  <NavigationProperty Name="Superordinate"
                      Type="SalesModel.SalesOrganization"
Nullable="false" />
</EntityType>
<EntityType Name="SalesOrganization">
  <Key>
    <PropertyRef Name="ID" />
  </Key>
  <Property Name="ID" Type="Edm.String" Nullable="false" />
  <Property Name="Name" Type="Edm.String" />
  <NavigationProperty Name="Relations"

Type="Collection(SalesModel.SalesOrganizationRelation)"
                      Nullable="false" ContainsTarget="true"
/>
  <Annotation Term="Aggregation.RecursiveHierarchy"
              Qualifier="MultiParentHierarchy">
    <Record>
      <PropertyValue Property="NodeProperty"
                     PropertyPath="ID" />
      <PropertyValue Property="ParentNavigationProperty"

NavigationPropertyPath="Relations/Superordinate" />
    </Record>
  </Annotation>
</EntityType>
```

Further assume the following relationships between sales organizations:

ID	Relations/SuperordinateID	Relations/Weight
US	Sales	1
EMEA	Sales	1
EMEA Central	EMEA	1
Atlantis	US	0.6
Atlantis	EMEA	0.4
Phobos	Mars	1

Then Atlantis is a node with two parents. The standard hierarchical transformations disregard the weight property and consider both parents equally valid (but see example 118).

In a traversal with start node Sales only:

```
GET /service/SalesOrganizations?$apply=

traverse($root/SalesOrganizations,MultiParentHierarchy,ID,pr
eorder,
             filter(ID eq 'Sales'))
```

Mars and Phobos cannot be reached and hence are orphans. But they can be made descendants of the start node Sales by adding a relationship. Note the collection-valued segment of the ParentNavigationProperty appears at the end of the resource path and the subsequent single-valued segment appears in the payload:

```
POST /service/SalesOrganizations('Mars')/Relations
Content-Type: application/json

{ "Superordinate": { "@id": "SalesOrganizations('Sales')" }
}
```

Since this example contains no referential constraint, there is no analogy to example 116. The alias SuperordinateID cannot be used in the payload, the following request is invalid:

```
POST /service/SalesOrganizations('Mars')/Relations
Content-Type: application/json

{ "SuperordinateID": "Sales" }
```

The alias SuperordinateID is used in the request to delete the added relationship again:

```
DELETE
/service/SalesOrganizations('Mars')/Relations('Sales')
```

⚠ Example 118: Continuing example 117, assume a custom aggregate MultiParentWeightedTotal that computes the total sales amount weighted by the SalesOrganizationRelation/Weight properties along the @Aggregation.UpPath#MultiParentHierarchy of a sales organization:

```
<Annotations Target="SalesData.Sales">
  <Annotation Term="Aggregation.CustomAggregate"
    Qualifier="MultiParentWeightedTotal"
String="Edm.Decimal" />
</Annotations>
```

Then rolluprecursive can be used to aggregate the weighted sales amounts with the request below. The traverse transformation produces an output set \(H'\) in which sales organizations with multiple parents occur multiple times. For each occurrence \(x\) in \(H'\), the rolluprecursive algorithm determines a sales collection \(F(x)\) and the custom aggregate MultiParentWeightedTotal evaluates the path SalesOrganization/@Aggregation.UpPath#MultiParentHierarchy relative to that collection:

```
GET /service/Sales?$apply=groupby(
    (rolluprecursive(
      $root/SalesOrganizations,
      MultiParentHierarchy,
      SalesOrganization/ID,
      traverse(
        $root/SalesOrganizations,
        MultiParentHierarchy,
        SalesOrganization/ID,
        preorder))),
    aggregate(MultiParentWeightedTotal))
```

```
Assume that in addition to the sales in the example data
~~"CustomerCountryAverage": 9.5, "Currency": { "Code": "USD" }~~
~~    },~~
~~    { "@odata.id": null,~~
~~      "CustomerCountryAverage":   5, "Currency": { "Code": "EUR" }~~
~~    }~~
~~  ]~~
~~}~~
```

~~7.8 Transformation Sequences~~

there are sales of 10 in Atlantis. Then 60% of them would contribute to the US sales organization and 40% to the EMEA sales organization. Without the weights, all duplicate nodes would contribute the same aggregate result, therefore this example only makes sense in connection with a custom aggregate that considers the weights.

Note that rolluprecursive must preserve the preorder established by traverse:

```
{
  "@context":
"$metadata#Sales(SalesOrganization(),MultiParentWeightedTota
l)",
  "value": [
    { "SalesOrganization": { "ID": "Sales", "Name":
"Corporate Sales",
        "@Aggregation.UpPath#MultiParentHierarchy": [ ] },
      "MultiParentWeightedTotal": 34 },
    { "SalesOrganization": { "ID": "US", "Name": "US",
        "@Aggregation.UpPath#MultiParentHierarchy": [
"Sales" ] },
      "MultiParentWeightedTotal": 25 },
    { "SalesOrganization": { "ID": "Atlantis", "Name":
"Atlantis",
        "@Aggregation.UpPath#MultiParentHierarchy": [ "US",
"Sales" ] },
      "MultiParentWeightedTotal": 6 },
    ...
    { "SalesOrganization": { "ID": "EMEA", "Name": "EMEA",
        "@Aggregation.UpPath#MultiParentHierarchy": [
"Sales" ] },
      "MultiParentWeightedTotal": 9 },
    { "SalesOrganization": { "ID": "Atlantis", "Name":
"Atlantis",
        "@Aggregation.UpPath#MultiParentHierarchy": [
"EMEA", "Sales" ] },
      "MultiParentWeightedTotal": 4 },
    ...
  ]
}
```

7.11 Transformation Sequences

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

Example ~~:~~119:

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

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

```
{
  "@~~odata.~~context": "$metadata#Sales(Total)",
  "value": [
~~  { "@odata.id": null,~~
    { "Total@type": "Decimal", "Total": 2 }
  ]
}
```

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

Example ~~:~~120:

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

~~*results in*~~

```
~~{~~
```

~~"@odata.~~*results in*

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

Example ~~:~~121: Revisiting example 16~~the in section~~ for using the from keyword with the aggregate function, the request

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

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

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

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

Example ~~:~~122: getting the population per country with

```
GET
~~~/~~/service/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,~~@type": "Int32", "TotalPopulation":
1412000000 },
     { "Continent": { "Name": "Asia" }, "Country": { "Name":
"India" },~~"TotalPopulation": 390.600.000 },~~
~~  ...~~
~~]~~
      "TotalPopulation@type": "Int32", "TotalPopulation":
1408000000 },
     ...
  ]
}
```

Example ~~.~~123: all countries with megacities and their continents

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

Example ~~.~~124: all countries with tens of millions of city dwellers and the continents only for these countries

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

~~OR~~

or

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

```
                          aggregate(Population with sum as
CountryPopulation))
                   /filter(CountryPopulation ge 10000000)

/groupby((rollup(Continent/Name,Country/Name)),
                             aggregate(CountryPopulation
with sum
                                       as TotalPopulation))
```

Example ~~:~~125: all countries with tens of millions of city dwellers and all continents with cities independent of their size

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

Example ~~:~~126: assuming the data model includes a sales order entity set with related sets for order items and customers, the base set as well as the related items can be filtered before aggregation

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

Example 127: assuming that Amount is a custom aggregate in addition to the property, determine the total for countries with an Amount greater than 1000

```
GET /service/SalesOrders?$apply=
  groupby((Customer/Country),aggregate(Amount))
  /filter(Amount gt 1000)
  /aggregate(Amount)
```

~~8~~ Example 128~~Conformance~~

: The output set of the concat transformation contains Sales entities multiple times with conflicting related AugmentedProduct entities that cannot be aggregated by the second transformation.

```
GET /service/Sales?$apply=
  concat(addnested(Product,compute(0.1 as Discount) as
AugmentedProduct),
         addnested(Product,compute(0.2 as Discount) as
AugmentedProduct))
  /aggregate(AugmentedProduct/Discount with max as
MaxDiscount)
```

results in an error.

Example 129: The nest transformation can be used inside groupby to produce one or more collection-valued properties per group.

```
GET /service/Sales?$apply=groupby((Product/Category/ID),
                      nest(groupby((Customer/ID)) as
Customers))
```

results in

```
{

"@context":"$metadata#Sales(Product(Category(ID)),Customers(
))",
  "value": [
    { "Product": { "Category": { "ID": "PG1" } },
      "Customers@context": "#Sales(Customer(ID))",
      "Customers": [ { "Customer": { "ID": "C1" } },
                     { "Customer": { "ID": "C2" } },
                     { "Customer": { "ID": "C3" } } ] },
    { "Product": { "Category": { "ID": "PG2" } },
      "Customers@context": "#Sales(Customer(ID))",
      "Customers": [ { "Customer": { "ID": "C1" } },
                     { "Customer": { "ID": "C2" } },
                     { "Customer": { "ID": "C3" } } ] }
  ]
}
```

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 annotation ApplySupported~~.~~.

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. References

This appendix contains the normative references that are used in this document.

While any hyperlinks included in this appendix were valid at the time of publication, OASIS cannot guarantee their long-term validity.

A.1 Normative References

The following documents are referenced in such a way that some or all of their content constitutes requirements of this document.

[OData-ABNF]

ABNF components: OData ABNF Construction Rules Version 4.01 and OData ABNF Test Cases.
See link in "Related work" section on cover page.

[OData-Agg-ABNF]

OData Aggregation ABNF Construction Rules Version 4.0.
See link in "Additional artifacts" section on cover page.

[OData-CSDL]

OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01.
See link in "Related work" section on cover page.

OData Common Schema Definition Language (CSDL) XML Representation Version 4.01.
See link in "Related work" section on cover page.

[OData-JSON]

OData JSON Format Version 4.01.
See link in "Related work" section on cover page.

[OData-Protocol]

OData Version 4.01. Part 1: Protocol.
See link in "Related work" section on cover page.

[OData-URL]

OData Version 4.01. Part 2: URL Conventions.
See link in "Related work" section on cover page.

[OData-VocAggr]

OData Aggregation Vocabulary.
See link in "Additional artifacts" section on cover page.

[OData-VocCore]

OData Core Vocabulary.
See link in "Related work" section on cover page.

[RFC2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997
https://www.rfc-editor.org/info/rfc2119.

[RFC8174]

~~Appendix A.~~ *Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017*
https://www.rfc-editor.org/info/rfc8174~~Acknowledgments~~

.

Appendix B. Acknowledgments

B.1 Special Thanks

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

~~Appendix B.~~ B.2 Participants~~Revision History~~

OData TC Members:

First Name	Last Name	Company
George	Ericson	Dell
Hubert	Heijkers	IBM
Ling	Jin	IBM
Stefan	Hagen	Individual
Michael	Pizzo	Microsoft
Christof	Sprenger	Microsoft
Ralf	Handl	SAP SE
Gerald	Krause	SAP SE
Heiko	Theißen	SAP SE
Martin	Zurmühl	SAP SE

Appendix C. Revision History

Revision	Date	Editor	Changes Made
Working Draft 01	2012-11-12	Ralf Handl	Translated contribution into OASIS format

Revision	Date	Editor	Changes Made
Committee Specification Draft 01	2013-07-25	Ralf Handl Hubert Heijkers Gerald Krause Michael Pizzo Martin Zurmuehl	Switched to pipe-and-filter-style query language based on composable set transformations Fleshed out examples and addressed numerous editorial and technical issues processed through the TC Added Conformance section
Committee Specification Draft 02	2014-01-09	Ralf Handl Hubert Heijkers Gerald Krause Michael Pizzo Martin Zurmuehl	Dynamic properties used all aggregated values~~.~~ either via aliases or via custom aggregates Refactored annotations
Committee Specification Draft 03	2015-07-16	Ralf Handl Hubert Heijkers Gerald Krause Michael Pizzo Martin Zurmuehl	Added compute transformation Minor clean-up
Committee Specification Draft 04	2023-07-05	Ralf Handl Hubert Heijkers Gerald Krause Michael Pizzo Heiko Theißen	Added section about fundamentals of input and output sets Algorithmic descriptions of transformations Added join and outerjoin transformations, replaced expand by addnested Added transformations orderby, skip, top, nest Added transformations for recursive hierarchies, updated related filter

Revision	Date	Editor	Changes Made
			functions Added functions evaluable on a collection, introduced keyword $these Merged section 4 "Representation of Aggregated Instances" into section 3 Remove actions and functions (except set transformations) on aggregated entities, adapted section "Actions and Functions on Aggregated Entities"

Appendix D. Notices

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specification, Candidate OASIS Standard, OASIS Standard, or Approved Errata).

[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 Standards Final Deliverable, 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 deliverable.]

[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 OASIS Standards Final Deliverable 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 OASIS Standards Final Deliverable. 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 OASIS Standards Final Deliverable 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 Standards Final Deliverable, 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 https://www.oasis-open.org/policies-guidelines/trademark/ for above guidance.