OData Extension for Data Aggregation
A Directional White Paper

1 Introduction

This paper documents some use cases, initial requirements, examples and design principles for an OData extension for data aggregation. It is non-normative and is intended to seed discussion in the OASIS OData TC for the development of an OASIS standard OData extension defining the representation and semantics for aggregation of data supporting multidimensional modeling.

We want to add the notion of aggregation to OData, without changing any of the base principles in OData. In the following sections we outline a representation and semantics for aggregation of data supporting multidimensional modeling, especially:

- Annotate entity sets and/or entity types with annotations representing analytic concepts such as dimensions, hierarchies, measures and key performance indicators
- Define semantics and operations for querying aggregated data
- Define results format for queries containing aggregated data

Note: If you as a reader are hoping to find anything remotely related to "advanced analytics" capabilities added to OData then you are unfortunately reading the wrong document. In this context think about augmenting the Entity Data Model (EDM) with annotations adding analytic type context to the model and some basic grouping/aggregation functionality (min/max/sum/average/count/distinct-count, provider specific and rule based aggregations) based on the data exposed in the EDM.

1.1 Status

Version 1.0 2012-05-18

1.2 Authors

Ralf Handl, SAP
Siva Harinath, Microsoft
Hubert Heijkers, IBM
Gerald Krause, SAP
Mike Pizzo, Microsoft

1.3 Background

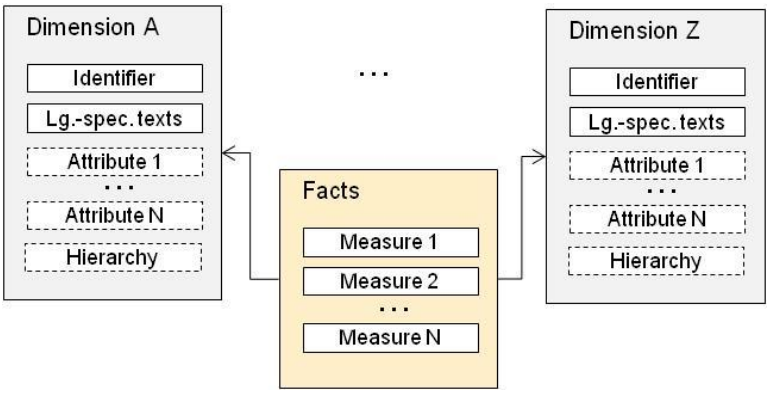
OData services expose a data model that describes the schema of the service in terms of the Entity Data Model (EDM), an Entity-Relationship model that describes the data and then allows for querying that data. The responses returned by an OData provider are based on that exposed data model and retain the relationships between the entities in the model. Adding the notion of aggregation to OData, without changing any of the base principles in OData as is, has two sides to it:

1. Means for the server to describe an "analytical shape" of the data represented by the service
2. Means for the client to query an "analytical shape" on top of any given data model (for sufficiently capable servers/services)

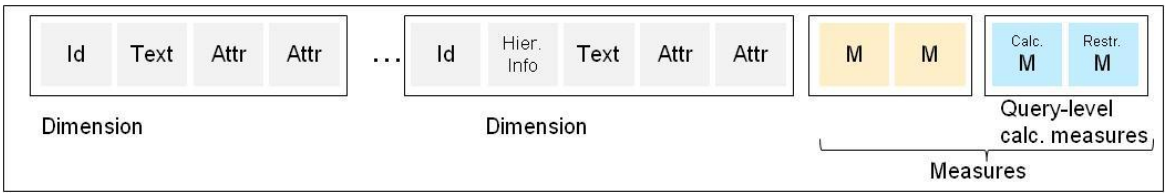
It's important to notice that, while each of these two sides might be valuable in its own right and can be used independently of the other, their combination provides additional value for clients. The descriptions provided by the server will help a consumer understand more of the data structure looking at the service's exposed data model from an analytics perspective, whereas the query extensions allow the clients to express an "analytical shape" for a particular query. The query extensions will also allow clients to refer to the server-described "analytical shape" as shorthand.

Multi-dimensional data – List of "facts" containing "measures" (= values that can be aggregated in the given dimensional context) and "dimensions" (= descriptive values that cannot be aggregated).
Dimension – Query element used for creating perspectives on data via drill-down and slicing.
Dimension members – Finite set of classifications representing the possible instances of a dimension. A member has an ID, an associated text, and attributes.
Dimension hierarchy – Organizes dimension members in a tree-like structure. Members are tree nodes at various levels.
Measures – Facts for a combination of dimension members. Facts are key figure values with a defined aggregation behavior, e.g. summation, average, minimum, maximum, etc.

The following diagram shows the relationship between Dimensions and Measures in a hierarchical star-like schema model.



Both query extensions and descriptive annotations can be applied to star-like schemas as well as partly or fully denormalized schemas.



Note that OData's EDM does not imply a storage model; it may be a completely conceptual model whose data shape is calculated on-the-fly for each request. The actual "entity-relationship shape" of the model, and consequently the "analytical shape" described by the annotations sketched here, should be chosen to simplify understanding and querying data by the target audience of a service. Different target audiences may well require differently shaped services on top of the same storage model: beauty is in the eye of the beholder.

1.4 Motivation

OData represents data as RESTful resources that make it easy for clients to "browse" through this "web of data", following relations between data elements ("entities") that are exposed as hyperlinks to other web resources. In addition to this hypermedia-driven data access, OData offers query capabilities via a small number of features (filtering, paging, projecting, and expanding along associations) that are by themselves intentionally simple and can be freely combined into an astonishingly powerful language.

Adding simple aggregation capabilities to the mix of query features avoids cluttering OData services with an exponential number of explicitly modeled "aggregation level entities" or else restricting the client to a small subset of predefined aggregations.

2 Requirements to Fulfill for Supporting Data Aggregation Scenarios in OData

The following capabilities must be supported in order to work with aggregated data in OData:

- Specify a set of entities (entries) that are to be grouped into subsets and retrieved as a single entity per subset representing the aggregate of that subset
- Specify how these groups are formed
- Specify how these groups are aggregated
- Describe how results of aggregating requests are represented in OData
- Aggregate entities are normal OData entities. In particular;
    - They can be individually identified and addressed
    - They can link to other entities
        - "Inherited" links from the group they represent, and that still are meaningful for the aggregate, or
        - New links to "drill" to sets of less aggregated entities

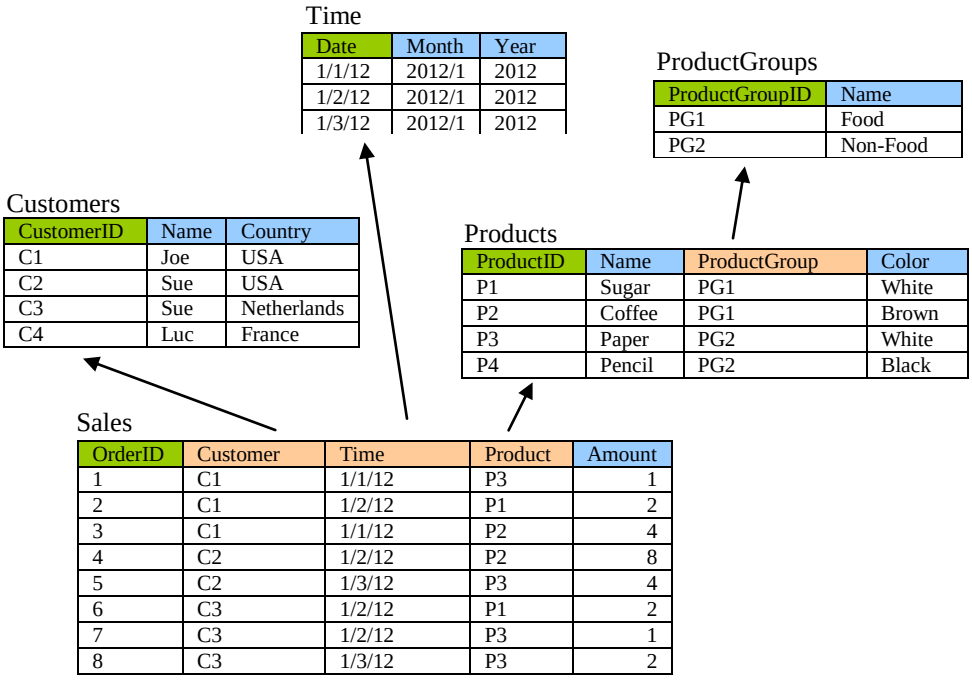
    - They can link to actions and/or function that can be invoked on them
- Describe "meaning" of OData resources in analytic terms

3 Examples

The following examples describe possible annotations and extensions to OData to support data aggregation. Although concrete annotations, functions, and behavior are described, they are intended to be purely illustrative and not prescriptive.

Let's look at a simple data model and project some analytic/OLAP query on top of that.

The following is the schema that we'll use for these samples that follow. It merely represents some sales of products for a set of customers:



Time

Date	Month	Year
1/1/12	2012/1	2012
1/2/12	2012/1	2012
1/3/12	2012/1	2012

ProductGroups

ProductGroupID	Name
PG1	Food
PG2	Non-Food

Customers

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

Products

ProductID	Name	ProductGroup	Color
P1	Sugar	PG1	White
P2	Coffee	PG1	Brown
P3	Paper	PG2	White
P4	Pencil	PG2	Black

Sales

OrderID	Customer	Time	Product	Amount
1	C1	1/1/12	P3	1
2	C1	1/2/12	P1	2
3	C1	1/1/12	P2	4
4	C2	1/2/12	P2	8
5	C2	1/3/12	P3	4
6	C3	1/2/12	P1	2
7	C3	1/2/12	P3	1
8	C3	1/3/12	P3	2

Note that the columns marked in orange are navigation properties whereas the columns marked in green are identifying the properties that make up the key for those entities. The values shown the in orange columns represent the links to the respective navigation target.

It would be pretty easy to envision projecting an "analytical shape", a cube if you will, on top of this data model, consisting of a "Sales" cube with one measure called "Amount". Since this measure represents individual sales values, unlike a balance or count of any sort, the default aggregation behavior would presumably be summation in this case. The three dimensions/axes for which data is specified here are:

- Product, with a hierarchy based on Product Group and Product
- Customer, with a hierarchy based on Country and Customer
- Time, with a hierarchy based on Year, Month and Date

To help the consumer of this service, the $metadata is accordingly annotated using the vocabulary sketched in section “5.1 Annotations”.

From this service we might like to retrieve the cross-table:

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

A flattened view of this table might look like the following:

Customer/Country	Customer/Name	Product/ProductGroup/Name	Product/Name	Amount
USA	Joe	Non-Food	Paper	1
USA	Joe	Food	Sugar	2
USA	Joe	Food	Coffee	4
USA	Sue	Food	Coffee	8
USA	Sue	Non-Food	Paper	4
Netherlands	Sue	Food	Sugar	2
Netherlands	Sue	Non-Food	Paper	3
USA	**NULL**	**Food**	**Sugar**	**2**
USA	**NULL**	**Food**	**Coffee**	**12**
USA	**NULL**	**Non-Food**	**Paper**	**5**
Netherlands	**NULL**	**Food**	**Sugar**	**2**
Netherlands	**NULL**	**Non-Food**	**Paper**	**1**
USA	**Joe**	**Food**	**NULL**	**6**
USA	**Joe**	**Non-Food**	**NULL**	**1**
USA	**Sue**	**Food**	**NULL**	**8**
USA	**Sue**	**Non-Food**	**NULL**	**4**
Netherlands	**Sue**	**Food**	**NULL**	**2**
Netherlands	**Sue**	**Non-Food**	**NULL**	**3**
USA	**NULL**	**Food**	**NULL**	**14**
USA	**NULL**	**Non-Food**	**NULL**	**5**
Netherlands	**NULL**	**Food**	**NULL**	**2**
Netherlands	**NULL**	**Non-Food**	**NULL**	**3**

Note that this result contains seven fully qualified aggregate values and fifteen rollup, or “subtotal”, rows (shown in bold).

However, a standard OData request:

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

would instead retrieve only the unaggregated base values:

Customer/County	Customer/Name	Product/ProductGroup/Name	Product/Name	Amount
USA	Joe	Non-Food	Paper	1
USA	Joe	Food	Sugar	2
USA	Joe	Food	Coffee	4
USA	Sue	Food	Coffee	8
USA	Sue	Non-Food	Paper	4
Netherlands	Sue	Food	Sugar	2
Netherlands	Sue	Non-Food	Paper	1
Netherlands	Sue	Non-Food	Paper	2

Note that the response in OData is not exactly a flat table, but slightly "folded" according to the relationships expressed in the data model exposed by the service. Using a simplified JSON format, it might look like:

```
[
  { Customer: { Country: "USA", Name: "Joe" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 1
  },
  { Customer: { Country: "USA", Name: "Joe" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Sugar" },
    Amount: 2
  },
  { Customer: { Country: "USA", Name: "Joe" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Coffee" },
    Amount: 4
  },
  { Customer: { Country: "USA", Name: "Sue" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Coffee" },
    Amount: 8
  },
  { Customer: { Country: "USA", Name: "Sue" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 4
  },
  { Customer: { Country: "Netherlands", Name: "Sue" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Sugar" },
    Amount: 2
  },
  { Customer: { Country: "Netherlands", Name: "Sue" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 1
  },
  { Customer: { Country: "Netherlands", Name: "Sue" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 2
  }
]
```

Actually we'd like to get only one aggregated row instead of the last two as those both provide data for the same cell in the cross-table we are trying to build. So let's invent a new query option to tell the server that it should aggregate the amount:

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

Note that we've introduced several conventions here:

- We tell the server only which measures to aggregate; the server applies the appropriate (default if you will) aggregation function (summation in this case).
- The projection chosen with the $select now defines the aggregation context, i.e. the server will group by those properties in $select that are the result of the aggregation requested in $aggregate.

The results of this request would produce only the seven (aggregated) rows:

Customer/Country	Customer/Name	Product/ProductGroup/Name	Product/Name	Amount
USA	Joe	Non-Food	Paper	1
USA	Joe	Food	Sugar	2
USA	Joe	Food	Coffee	4
USA	Sue	Food	Coffee	8
USA	Sue	Non-Food	Paper	4
Netherlands	Sue	Food	Sugar	2
Netherlands	Sue	Non-Food	Paper	**3**

Or, in simplified JSON:

```
[
  { Customer: { Country: "USA", Name: "Joe" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 1
  },
  { Customer: { Country: "USA", Name: "Joe" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Sugar" },
    Amount: 2
  },
  { Customer: { Country: "USA", Name: "Joe" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Coffee" },
    Amount: 4
  },
  { Customer: { Country: "USA", Name: "Sue" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Coffee" },
    Amount: 8
  },
  { Customer: { Country: "USA", Name: "Sue" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 4
  },
  { Customer: { Country: "Netherlands", Name: "Sue" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Sugar" },
    Amount: 2
  },
  { Customer: { Country: "Netherlands", Name: "Sue" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 3
  }
]
```

Note that each of these rows has the same structure as an individual row, but the values are the aggregated values across the different dimension properties. In particular, note that the row representing the sale of Paper by Sue from the Netherlands is an aggregated value summing the Amounts from two rows in the source table.

Aggregated rows have the same structure as the individual rows, so the shape of the results can still mirror the shape described by the service. However, aggregated rows have different ids and self links than unaggregated rows. An aggregated row's self link must encode the necessary information to re-retrieve that particular aggregate value, for instance the set of unique dimension property values that the aggregate represents.

To produce the missing fifteen subtotal rows we introduce a $rollup query option:

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

```
(Product/ProductGroup/Name,Product/Name)
```

The $rollup query option is what specifies the "analytical shape", the hierarchies the client is interested in, for this query. Adding the $rollup results in the addition of those fifteen aggregated values which make up all intersections for Customer/Country and Product/ProductGroup/Name in this example.

```
[
  { Customer: { Country: "USA" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Sugar" },
    Amount: 2
  },
  { Customer: { Country: "USA" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Coffee" },
    Amount: 12
  },
  { Customer: { Country: "USA" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 5
  },
  { Customer: { Country: "Netherlands" },
    Product: { ProductGroup: { Name: "Food" }, Name: "Sugar" },
    Amount: 2
  },
  { Customer: { Country: "Netherlands" },
    Product: { ProductGroup: { Name: "Non-Food" }, Name: "Paper" },
    Amount: 1
  },
  { Customer: { Country: "USA", Name: "Joe" },
    Product: { ProductGroup: { Name: "Food" } },
    Amount: 6
  },
  ...
  { Customer: { Country: "USA" },
    Product: { ProductGroup: { Name: "Food" } },
    Amount: 14
  },
  ...
]
```

The properties that are aggregated away are omitted from the response payload.

Note that all properties referenced in the $rollup clause must be part of the $select clause but that the $select clause might contain more properties by which we still group and that provide "context" for the aggregations being returned.

4 Design Principles

OData is an application-level protocol for interacting with data via RESTful web services. An OData service's contract is defined by simple, well-defined conventions and semantics applied to the data model exposed by the service, providing a high level of semantic interoperability between loosely coupled clients and services.

The design principles of OData are to:

- Make it easy to implement and consume a basic OData service over a variety of data sources. Rather than try and expose the full functionality of all stores, define

common features for core data retrieval and update scenarios and incremental, optional features for more advanced scenarios.
- Leverage Data Models to guide clients through common interaction patterns rather than force clients to write complex queries against raw data
- Define consistency across the protocol and a single way to express each piece of functionality

The design principles of OData extensions are to:
- Ensure extensions do not violate the core semantics of OData
- Avoid defining different representations for common concepts across extensions
- Ensure independent extensions compose well
- Ensure clients can ignore extended functionality and still query and consume data correctly

Extending OData to support Data Aggregation should follow the following design principles:
- Extensions for data aggregation should not break existing clients and client libraries; existing clients should be able to consume models containing predefined measures and annotations without understanding those annotations or additional semantics
- Clients should trigger aggregation explicitly; unless the client does something different it should get existing OData behavior.
- The shape of the result should follow the shape of the model: don't break OData's type system
- Aggregate rows should have self-links and unique ids
- Supported aggregation behavior should be described via metadata annotations
- Client should retain full control over aggregation behavior
- Server should exhibit the most useful / minimally astonishing default behavior
- Extensions should work on "star" schemas as well as "flattened" (denormalized) schemas
- Extensions should work with "analytical" as well as "tabular" data providers

5 Technical Direction

This section attempts to formalize a proposed extension syntax and describe its interpretation in a "define by example" fashion. It is far from complete, and in some cases alternative interpretations have been stated as a starting point for further discussions.

5.1 Annotations

The following annotations are added to describe analytic aspects of an entity model.

5.1.1 Measures

Measures are identified using the Measure annotation term. This term may be applied to any property name that can be used in $aggregate.

```
<!--Measures define what can be used in a $aggregate-->
<ComplexType Name="Measure" Abstract="true"/>
```

A measure may be explicitly associated with an existing property through a default aggregation. This measure has the same name as the dependent property and, if specified in $aggregate, must be aliased in order to refer separately to the unaggregated property elsewhere in the request. Such a dependent measure may specify a predefined default aggregation function of "sum", "average", "max", "min", or some service-defined custom aggregation.

```
<ComplexType Name="DependentMeasure" Base="Self.Measure">
  <TypeAnnotation Term="Vocabulary.AppliesToProperty"/>
  <Property Name="DefaultAggregrationFunction"
            Type="Edm.String" Nullable="true"/>
  <Property Name="AcceptedAggregrationFunctions"
            Type="Collection(Edm.String)" Nullable="true"/>
</ComplexType>
```

A measure that cannot be specified in $select without being included in $aggregate is not exposed as an entity property, but rather described through a "Floating Aggregate" annotation. Since floating aggregates are not exposed as properties on the entity type, they are generally treated as dynamic properties.

```
<ComplexType Name="FloatingMeasure" Base="Self.Measure">
  <Property Name="Name" Type="Edm.String" />
  <Property Name="Type" Type="Edm.String" />
  <Property Name="InputProperties" Type="Collection(Edm.String)"/>
</ComplexType>
```

The set of floating measures is represented through a "Measures" annotation term that may be associated with an entity set or an entity container that supports querying.

```
<ValueAnnotation Name="Measures" Type="Collection(Self.FloatingMeasure)"/>
```

5.1.2 Hierarchies

A group of properties can form a hierarchy:

```
<ComplexType Name="LeveledHierarchy">
  <Property Name="Name" Type="String" Nullable="false"/>
  <!-- Ordered list of properties in the hierarchy -->
  <Property Name="Levels" Type="Collection(Edm.String)" Nullable="false">
    <TypeAnnotation Term="Vocabulary.ReferencesProperty"/>
  </Property>
</ComplexType>

<ComplexType Name="RecursiveHierarchy">
  <Property Name="HierarchyNodeIDProperty" Type="Edm.String"
            Nullable="false">
    <TypeAnnotation Term="Vocabulary.ReferencesProperty"/>
  </Property>
```

```
  <Property Name="HierarchyParentNodeIDProperty" Type="Edm.String"
            Nullable="false">
    <TypeAnnotation Term="Vocabulary.ReferencesProperty"/>
  </Property>
  <Property Name="HierarchyLevelProperty" Type="Edm.String">
    <TypeAnnotation Term="Vocabulary.ReferencesProperty"/>
  </Property>
</ComplexType>
```

These terms are applied to the Sales entity type, so that they can be used by clients for requesting additional aggregation levels, see section $rollup:

```
<EntityType Name="Sales">
  <Key>
    <PropertyRef Name="OrderID" />
  </Key>
  <Property Name="OrderID" Type="Edm.Int32" Nullable="false" />
  <Property Name="Amount" Type="Edm.Decimal" Nullable="false"
            Precision="5" Scale="2">
    <TypeAnnotation Term="DataAggregation.DependentMeasure">
      <PropertyValue Property="DefaultAggregationFunction" String="sum">
    </TypeAnnotation>
  </Property>
  <NavigationProperty Name="Product" Relationship="Model1.ProductSales"
                      ToRole="Product" FromRole="Sales" />
  <NavigationProperty Name="Customer" Relationship="Model1.CustomerSales"
                      ToRole="Customer" FromRole="Sales" />
  <NavigationProperty Name="Time" Relationship="Model1.SalesTime"
                      ToRole="Time" FromRole="Sales" />
</EntityType>

<EntityType Name="Product">
  <Key>
    <PropertyRef Name="ProductID" />
  </Key>
  <Property Name="ProductID" Type="Edm.String" Nullable="false" />
  <Property Name="Name" Type="Edm.String" Nullable="false" />
  <Property Name="Color" Type="Edm.String" Nullable="false" />
  <NavigationProperty Name="ProductGroup"
                      Relationship="Model1.ProductGroupProduct"
                      ToRole="ProductGroup" FromRole="Product" />
  <NavigationProperty Name="Sales" Relationship="Model1.ProductSales"
                      ToRole="Sales" FromRole="Product" />
  <TypeAnnotation Term="DataAggregation.LeveledHierarchy">
    <PropertyValue Property="Name" String="ProductHierarchy"/>
    <PropertyValue Property="Levels">
      <Collection>
        <String="ProductGroup/Name"/>
        <String="Name"/>
      </Collection>
    </PropertyValue>
  </TypeAnnotation>
</EntityType>
```

5.1.3 Functions and Actions on Aggregated Entities

Functions and actions with a binding parameter may or may not be applicable to an aggregated entity. By default we assume such bindings are not applicable to aggregated entities, and define a term to annotate those functions/actions that are also applicable to (a subset of the) aggregated entities. The applicability most likely will depend on the aggregation level, so these functions/actions must not be "always bindable". Assume the product is an implicit input for a function bindable to Sales, then aggregating away the product makes this function inapplicable.

```
<ComplexType Name="AvailableOnAggregates" BaseType="odata.core.Tag">
  <Property Name="DependsOnProperties" Type="Collection(String)"
            Nullable="true">
    <TypeAnnotation Term="Vocabulary.ReferencesProperty"/>
  </Property>
</ComplexType>
```

5.1.4 Dimensions

If an entity set represents a denormalized schema with multiple dimensions, the server may want to tell the client which properties are grouped together, and which of them form the "dimension key":

```
<ComplexType Name="Dimension">
  <Property Name="Key" Type="Collection(String)">
    <TypeAnnotation Term="Vocabulary.ReferencesProperty"/>
  </Property>
  <Property Name="Properties" Type="Collection(String)">
    <TypeAnnotation Term="Vocabulary.ReferencesProperty"/>
  </Property>
</ComplexType>
```

5.2 Query Options

5.2.1 $aggregate

Aggregation behavior is triggered using the new query option $aggregate.

In the presence of $aggregate the system query option $select not only defines the shape of the result set, it also defines the "scope" of the aggregation. As in standard OData, $expand must be used to bring into scope properties from any related entities. Specifying properties from related entities whose navigation paths are not included in $expand is an error.

Without parameters $aggregate returns the distinct value combinations of properties listed in the $select system query option.

```
GET ~/Sales?$select=Customer/Name
           &$expand=Customer
           &$aggregate
```

will return

```
[
  { Customer: { Name: "Joe" } },
  { Customer: { Name: "Sue" } }
]
```

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

Note also that "Sue" appears only once although the customer base contains two different Sues. Including properties that guarantee the right level of uniqueness in the $select clause, CustomerID for example in this case, will repair that, so be careful what you ask for.

$aggregate can take a comma-separated list of property names, similar to $select (i.e. including navigation path segments). The listed properties will be part of the result and must be included in $select. Their values will be calculated using the server-defined default aggregation function within the $selected context, and they will not be considered for grouping.

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

will return

```
[
  { Customer: { Country: "Netherlands" }, Product: { Name: "Paper" }, Amount: 3 },
  { Customer: { Country: "Netherlands" }, Product: { Name: "Sugar" }, Amount: 2 },
  { Customer: { Country: "USA" }, Product: { Name: "Coffee" }, Amount: 12 },
  { Customer: { Country: "USA" }, Product: { Name: "Paper"  }, Amount:  5 },
  { Customer: { Country: "USA" }, Product: { Name: "Sugar"  }, Amount:  2 }
]
```

The shorthand $aggregate=* means: aggregate all properties that have been annotated as Measure and have a declared default aggregation function. In our example only the Amount property has been annotated as a Measure, and its default aggregation function is "sum", so the above request could also have been issued as

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

This shorthand is useful if the model represents a cube with strict separation of measures and dimensions, as it does what consumers of that cube would expect. In this kind of models measures will have a default aggregation function. If a measures listed in $select does not have a default aggregation function, it will not be considered part of the $aggregate list and instead be used for grouping.

Using $aggregate with no parameters is possible without $select, but the $aggregate will have no effect on the result as each row is uniquely identified by its key value(s).

Instead of using the server-defined default aggregation function, the client may specify one of the predefined aggregation functions min, max, sum, average, count, distinctCount, or a provider-specific function that is qualified with a namespace prefix.

- min, max, sum, and average take the name of a numeric property (optionally with path prefix) as argument. The result property will have the same type as the argument property if used without an alias name.
- count and distinctCount take the name of a simple or complex property (optionally with path prefix) as argument. The result property will have type Edm.Int64, and it must be given an alias name.

When applying an aggregation function to a property, an alias name may be provided to allow multiple aggregates for a single property. The alias will introduce a dynamic property in the entity type of the aggregated property used as the function argument to preserve the result shape. The alias must be a SimpleIdentifier (see [CSDL, section 2.2.6]), and it must not collide with names of declared properties of that entity type, nor with other aliases for properties of the same entity type. The alias must appear in the $select clause with the same path prefix as the aggregated property.

```
GET ~/Sales?$select=Customer/Country,Amount,AvgAm
           &$expand=Customer
           &$aggregate=sum(Amount),average(Amount) as AvgAm
```

will return

```
[
  { Customer: { Country: "Netherlands" }, Amount:  5, AvgAm: 2.5 },
  { Customer: { Country: "USA"         }, Amount: 19, AvgAm: 3.8 }
]
```

and

```
GET ~/Products?$select=Name,Sales/Amount,Sales/AvgAmt
              &$expand=Sales
              &$aggregate=sum(Sales/Amount),
                          avg(Sales/Amount) as AvgAmt
```

will return

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

Note that aggregation does not alter the cardinality of the Sales navigation property, and that it always returns a one-element array with an object containing all selected properties

of Sales, even if there were no “base” entities to be aggregated. This fact is instead expressed by all properties having NULL values, which makes the result easier to understand.

The aggregation function count(.) takes the name of a property as its argument. It counts the non-NULL values of this property. To count the entities in the group to be aggregated into a single entity count(*) can be used.

As with all non-default aggregations, count must always specify an alias.

```
GET ~/Sales?$select=Product/Name,SalesCount
           &$expand=Product
           &$aggregate=count(*) as SalesCount
```

would return:

```
[
  { Product: { Name: “Coffee” }, SalesCount: 2 },
  { Product: { Name: “Paper”  }, SalesCount: 4 },
  { Product: { Name: “Sugar”  }, SalesCount: 2 }
]
```

You can specify a navigation property (with path if necessary) to count the number of related entities:

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

would return:

```
[
  { Name: “Coffee”, SalesCount: 2},
  { Name: “Paper”,  SalesCount: 4},
  { Name: “Pencil”, SalesCount: 0},
  { Name: “Sugar”,  SalesCount: 2}
]
```

Note that, when specifying a navigation path, the count appears on the object containing the final navigation property (i.e., the parent of the entities being counted):

```
GET ~/ProductGroups?$select=Name,Products/SalesCount
              &$expand=Products
              &$aggregate=count(Products/Sales) as SalesCount
```

would return:

```
[
  { Name: “Food”, Products:[{SalesCount: 4}] },
  { Name: “Non-Food”, Products:[{SalesCount: 4}] }
]
```

The aggregation function distinctCount(.) takes the name of a property as its argument. It counts the distinct values of this property, omitting any NULL values. For navigation properties it counts the distinct entities in the union of all entities related to entities in the group:

```
GET ~/Customers?$select=Country,Sales/DistinctProducts
               &$expand=Sales
               &$aggregate=distinctCount(Sales/Product)
                           as DistinctProducts
```

returns the number of different products sold per country:

```
[
  { Country: "Netherlands", Sales: [{ DistinctProducts: 2 }] },
  { Country: "USA",         Sales: [{ DistinctProducts: 3 }] }
]
```

Note that DistinctProducts is located in the Sales entity as it is an aggregate of Sales/Product: as stated above aliasing just renames a property, it does not relocate it. Also note that associations are "expanded" in a left-outer-join fashion, starting from the target of the aggregation request, before grouping the entities for aggregation. Afterwards the results are "folded back" to match the cardinality:

```
GET ~/Customers?$select=Country,Sales/Product/Name
               &$expand=Sales,Sales/Product
               &$aggregate
```

returns the different products sold per country:

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

There's no hard distinction between "dimensions" and "measures": the same property can be aggregated and used to group the aggregated results:

```
GET ~/Sales?$select=Amount,TotalSales
           &$aggregate=sum(Amount) as TotalSales
```

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

```
[
  { Amount: 1, TotalSales: 2 },
  { Amount: 2, TotalSales: 6 },
  { Amount: 4, TotalSales: 8 },
  { Amount: 8, TotalSales: 8 }
]
```

5.2.2 $rollup

The $rollup query option is used to specify the "analytical shape"(the different levels of aggregation the client is interested in) for a particular query. Adding the $rollup query option results in adding additional entries to the result representing the aggregated values produced as a result of the rollup query option, in which progressively more properties, based on the specified named hierarchies or ad-hoc hierarchies (expressed as lists of properties), are omitted from those entities. Note that properties are grouped, using parentheses, to form a leveled hierarchy along which the aggregation needs to take place and that hierarchies are themselves comma separated. Aggregations will be provided for the cartesian product for the intersections along these hierarchies.

In the examples section we demonstrated the use of $rollup to retrieve those aggregated values that were required for the sample grid using the following request:

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

which results in additional entries representing the aggregated subtotals being returned. In this sample, at least the Customer/Name, the Product/Name or both have been ommitted.

An alternative shorthand using the sever-defined leveled hierarchy from the annotation example in the previous section would produce the same result

```
          &$rollup=(Customer/Country,Customer/Name),
                   ProductHierarchy
```

The hierarchy name is not enclosed in parentheses, so it can be distinguished from a one-level ad-hoc hierarchy using a property name that must be enclosed in parentheses.

Properties that have been "rolled up" are omitted from the response.

Note that $rollup stops one level earlier than GROUP BY ROLLUP in TSQL, see [TSQL ROLLUP]: per hierarchy the leftmost property is never rolled up. That's fine if the model contains a property for the "all" level (having only a single value). Otherwise the pseudo-property $all can be used to force rollup to the point where the leftmost "real" property is rolled up:

```
          &$rollup=($all,Customer/Country,Customer/Name)
```

will return two entities rolled up to country level, and one entity rolled up across all countries:

```
[
  ...
  { Customer: { Country: "Netherlands" },
    Product: { ProductGroup: { Name: "Food" } },
    Amount:  2
  },
  { Customer: { Country: "USA" },
    Product: { ProductGroup: { Name: "Food" } },
    Amount: 14
  },
  { Product: { ProductGroup: { Name: "Food" } },
    Amount: 16
  },
]
```

To rollup by the key of each related entity you can simply specify the name of the navigation property. When rolling up by key, all key fields for the related entity must be present in the $select list.

5.2.3 Filtering and Aggregation

$filter expressions need special treatment in conjunction with $aggregate.

Property names not appearing in $aggregate refer to unaggregated values. Alias names introduced in $aggregate via "as" can be used in $filter expressions and always refer to the values after aggregation. Property names appearing in $aggregate without an "as" alias also refer to values after aggregation. If all appearances of a property in $aggregate introduce an alias, the original property name may be used in $filter expressions and refers to values *before* aggregation.

This allows us to compute how much money we make with small sales:

```
GET ~/Sales?$select=TotalAmount
           &$aggregate=sum(Amount) as TotalAmount
           &$filter=Amount le 1
```

```
[
  { TotalAmount: 2 }
]
```

5.2.4 Queries Spanning Entity Sets

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

In some cases, however, requests may span entity sets with no predefined associations. Such queries could be facilitated by a general extension to OData that would allow requests to be rooted at the entity container, rather than an individual entity set. The entity container defines implicit navigation properties to each entity set (and potentially each function) it contains, and queries across entity sets could be supported by referring to properties qualified by entity set.

For example, if Customers and Countries were in separate entity sets with no defined relationship, to query all Customers for a particular country based on a common country code one could pose the following query:

```
GET ~SalesData?$select=Customers/Name,Countries/Name
        &$expand=Customers,Countries
        &$filter=
            (Customers/CountryCode eq Countries/CountryCode)
             And (Countries/Name eq 'USA')
```

would return:

```
[
       { Customers: [{Name: "Joe"}], Countries: [{Name: "USA"}] },
       { Customers: [{Name: "Sue"}], Countries: [{Name: "USA"}] }
]
```

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 clients to pose queries across entity sets based on other join conditions, such as relationships implied by a measure.

For example, the client could issue a query over the SalesData entity container:

```
GET ~SalesData?$select=Products/Name,Time/Date,Sales/Amount
               &$expand=Products,Time,Sales
               &$aggregate=Sales/Amount
```

Where the result would look like:

```
[
       { Sales:[{Amount:4}], Products:[{Name:"Sugar"}],Time:[{Date:1/2/12}] },
       { Sales:[{Amount:4}], Products:[{Name:"Coffee"}],Time:[{Date:1/1/12}] },
       { Sales:[{Amount:8}], Products:[{Name:"Coffee"}],Time:[{Date:1/2/12}] },
       { Sales:[{Amount:1}], Products:[{Name:"Paper"}],Time:[{Date:1/1/12}] },
       { Sales:[{Amount:1}], Products:[{Name:"Paper"}],Time:[{Date:1/2/12}] },
       { Sales:[{Amount:5}], Products:[{Name:"Paper"}],Time:[{Date:1/3/12}] },
]
```

The entity container may be annotated with measures that can be applied to aggregations from the entity container.

Applying such a term to the SalesData entity container for an "ActualOverSales" aggregate would look like:

```
<EntityContainer Name="SalesData" m:IsDefaultEntityContainer="true">
  <...>
  <TypeAnnotation Term="Measures">
    <Collection>
      <Record>
        <PropertyValue Property="Name" String="ActualOverSales" />
        <PropertyValue Property="Type" String="Edm.Integer" />
```

```
        </Record>
      </Collection>
    </TypeAnnotation>
  </EntityContainer>
```

The SalesData entity container would support the query:

```
GET ~SalesData?$select=Products/Name,Time/Month,
                          ActualOverSales
               &$expand=Products,Time
               &$aggregate=ActualOverSales
```

with the result:

```
[
       { ActualOverSales:10, Products:[{Name:"Sugar"}],Time:[{Month:"2012/1"}] },
       { ActualOverSales:-20, Products:[{Name:"Coffee"}],Time:[{Month:"2012/1"}] },
       { ActualOverSales:25, Products:[{Name:"Paper"}],Time:[{Month:"2012/1"}] },
]
```

6 Open questions, issues and work items

6.1 Alternate Designs

This section describes alternative options for some of the design choices described above.

6.1.1 Cross Joins

Cross-joins could be simulated by introducing an entity set that projected all properties of those entity sets that were allow to be cross-joined. This would allow arbitrary joins and aggregations across those entity sets. Whether the "cross-join" entity sets have associations to "dimension" entity sets (star-shaped models) or include the "dimension" properties (denormalized models) would be a matter of taste: both model types could be supported. We moved from this design to rooting cross-entityset queries at the EntityContainer because:

1) This aggregation entityset would have no meaningful key columns
2) The aggregation entityset would have no meaningful properties other than navigation properties to other entitysets
3) Defining associations from this entityset to all of the other entityset would pollute the model
4) We had a hard time defining a reasonable behavior for existing consumers querying this entityset
5) A useful feature, the ability to express queries across entitysets in general, came for free in the design when we rooted the queries at the EntityContainer.

6.1.2 Rollup Properties

Properties that are omitted from a response due to rolled-up could instead be exposed as null values in the response payload. In this case, an instance annotation could be used to specify that the property is null because of the $rollup.

```
{ Customer: { Country: "USA", Name: null },
  Product: { ProductGroup: { Name: "Food" }, Name: "Sugar" },
  Amount: 2,
  OData.Analytics.RolledUpProperties : { [ "Customer/Name" ] }
}
```

6.1.3 LeveledHierarchyLevel

We discussed defining a type term, LeveledHierarchyLevel, containing the level (as an integer) and the property reference to the property of that level. This doesn't rely on ordering of level properties in the Levels annotation, but requires defining a type for LeveledHierarchyLevel and then using a record constructor with PropertyValue elements in specifying the levels, rather than simply specifying the ordered names of property values.

6.2 Open Issues

6.2.1 Implicit $expand

Discuss whether the current OData behavior of having to explicitly $expand relations that have been used in $select can be in general relaxed to implicitly $expand those relations.

So we would be able to reference properties in $aggregate and $rollup which aren't explicitly listed $select but nevertheless are valid if $expand would bring them in scope. So $aggregate and/or $rollup would implicitly cause expansion of the included properties, as we were suggesting for $select.

6.2.2 Links for Aggregated Values

Discuss whether having links on aggregate entities that allow to "drill" to less aggregated values requires the entity type to be declared as OpenType="true". This could be avoided by using links that are not dynamic navigation properties, or by allowing dynamic navigation properties also on "non-open" types, i.e. requiring "open" only for adding data properties. Dynamic navigation properties have the added benefit of being $expandable.

6.2.3 Different Types of Hierarchies

Other types of hierarchies in addition to leveled and parent-child will have to be considered, and described via specialized annotations.
This needs to be carefully balanced to preserve the simplicity of OData.

6.2.4 Hierarchy-based Functions

Hierarchies may need additional hierarchy functions, e.g. for use in $filter expressions.
This needs to be carefully balanced to preserve the simplicity of OData.

6.2.5 Aggregation of Parent-Child Levels

Check if aggregation works as expected in parent-child levels. Questions being:

- Would one need to define explicitly that one is dealing with such a level.
- What is the value for the property being aggregated returns if the parent entity in a parent child relationship has data for that property too?

6.2.6 Predefined Aggregation Functions

We could define an enum for the set of known aggregation functions (rather than string), but this would require us to make the property optional or define a "custom" value in the enumeration and perhaps expose a separate property for services to describe custom aggregation functions.

```
<EnumType Name="AggregationFunction">
  <Member Name="sum" />
  <Member Name="max" />
  <Member Name="min" />
  <Member Name="average" />
  <Member Name="count" />
  <Member Name="distinctCount" />
  <Member Name="custom" />
</EnumType>
```

6.2.7 Additional Annotations

For the complete description of analytical resources, we will need further annotations to indicate amounts with related currencies/quantities with units, and text (caption) properties. In our previous discussions, we have outsourced these terms to other (more general) vocabularies. I would suggest to mention that such terms are also required, defined elsewhere (where?) and track such dependencies.

As not all (analytical) engines will offer the same capabilities, we will need annotations to e.g. express which properties can be used for grouping.

6.2.8 Filter Expressions on Aggregate

Consider if there are there filter expressions that cannot be evaluated at all due to their combined reference to before- and after-values? Do we need to worry? Servers can always respond with 400 Bad Request.

Adding $rollup to the mix raises the next bunch of questions:

- Is the filter applied before rollup, and then rollup takes place without further filtering?
- Or is the filter applied after rollup?

Example: filter for countries with small population, then rollup to continents: do we get all continents with small countries, or only continents with small population? I.e. just Antarctica for most definitions of “small”.

How is the aggregate value for rollup entities calculated: do we get the actual population per continent, or just the sum of the populations of the small countries?

What if the population is not just by country, but also by age class, and we filter for teenagers in countries with small population? Would we expect the continent rollup to reflect only teenagers, or all age groups?

6.2.9 Queries rooted on Entity Container

In order to support queries rooted at the entity container, we would need a way to differentiate between the name of the entity container and the name of an entity set within the container. This could be as simple as defining precedence; if the name of an entity set is the same as the name of its parent entity container, queries against that entity set must be qualified with the name of the entity container.

6.2.10 Defining the Return Type for an Entity Container

The structure of the Entity Container implicitly defines a type with navigation properties for each entity set. Should we define an explicit complex type for this in the model? This might be useful for tools that generate types for each EntityType/ComplexType in the model as the result could come back as that named type, but we would have to allow relationships on complex types (types without ids) as the entity container does not have a key. Alternatively we could define a name (or convention for the name) of an entity type associated with the Entity Container so that tools could generate strongly typed result objects based on the implicit properties, and services could serialize the top-level result as a specific type.

6.2.11 Set Functions for Complex Aggregations

A general extension to OData to add query options as composable functions to the path would allow greater composition. For example:

```
select distinct sales
  from (select city, sales from MySet group by city)
```

could be expressed as

```
GET ~/MySet/$Select(city,sales)/$Aggregate(sales)
           /$Select(sales)/$Aggregate()
```

Similar functions for $Filter(clause), $Aggregate(measures,rollup) would complete the mix.

6.2.12 Types for Aggregated Properties

Define rules for the types of dynamic properties created via aliasing, depending on the aggregation function and the type of the argument property.

6.3 References

[CSDL] http://www.odata.org/media/30001/[mc-csdl].pdf and http://www.odata.org/media/30002/OData CSDL Definition.html

[POLA] http://en.wikipedia.org/wiki/Principle_of_least_astonishment

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