100 XP

Aggregation pipeline

7 minutes

While many times you might need your query to return a set of documents to be processed programmatically with the collection's *find* method, other times you might just want the query to return an aggregation like a sum or an average. Azure Cosmos DB for MongoDB leverages on MongoDB's Aggregation pipelines that allow your queries to perform those aggregations at the server level. This gives you the ability to create more sophisticated queries.

Aggregation pipeline

Aggregation pipelines as the name suggests, uses a multi-stage data processing pipelines that transform and aggregate results. Let's take a look at some of the common stages used to define the pipelines.

- **$match** - Specifies the query conditions. Uses the same query format as the collection *find* method. Just like the *find* method, and if placed at the beginning of a pipeline, this stage uses the available indexes to improve performance.
- **$sort** - This stage sorts the results by one or more fields. If not preceded by the *$project*, *$unwind*, or *$group* stages, this stage also uses available indexes to improve performance.
- **$count** - Returns the number of documents passed from the previous stage.
- **$project** - Lists the fields to be returned and/or suppressed in the result set.
- **$unwind** - Deconstructs an array or subproperties. Be careful using *$unwind* with many documents, try filtering before the *$unwind* if necessary.
- **$group** - This stage groups documents based on the ***_id*** expression. This expression can range from one or more fields in your documents, to transformations from your documents or other preceding pipelines. Let' take a look at the **$group** pipeline in slightly more detail.

$group

While the *$match* and *$sort* pipelines filter and order our aggregations respectively, *$group* will not only define the grouping of the aggregation but also how to transform that grouping. The *$group* pipeline, will use different *accumulator operators* like *$sum* and *$avg* to transform our results. Let's take a look at some common accumulator operators used with this pipeline.

- **$first** - Returns a value from the first document for each group.
- **$last** - Returns a value from the last document for each group.
- **$min** - Returns the lowest expression value for each group.
- **$max** - Returns the highest expression value for each group.
- **$avg** - Returns an average of numerical values.
- **$sum** - Returns a sum of numerical values.

Using aggregation pipeline to query our data

Below we'll define some query scenarios to define some of our aggregation pipelines.

Let's assume we have a collection called *salesReceipts* and we have the following three documents in that collection.

JSON

```json
    {
        "_id": 1
        , "ReceiptNo": 1000
        , "SalesDetail": [
            {"ItemName":"Eggs","quantity":5,"priceperunit":3.55},
            {"ItemName":"Milk","quantity":2,"priceperunit":4.25},
            {"ItemName":"Bread","quantity":1,"priceperunit":1.33}
        ]
    }
```

JSON

```json
    {
        "_id": 2
        , "ReceiptNo": 1001
        , "SalesDetail": [
            {"ItemName":"Milk","quantity":1,"priceperunit":4.25},
            {"ItemName":"Eggs","quantity":3,"priceperunit":3.55},
            {"ItemName":"Flour","quantity":1,"priceperunit":2.45}
        ]
    }
```

JSON

```json
    {
        "_id": 3
        , "ReceiptNo": 1002
        , "SalesDetail": [
            {"ItemName":"Tomatoes","quantity":10,"priceperunit":0.35},
            {"ItemName":"Lettuce","quantity":1,"priceperunit":0.85},
            {"ItemName":"Onions","quantity":4,"priceperunit":0.57}
        ]
    }
```

Aggregation pipeline examples using the Mongo Shell

To run an aggregation pipeline query in the mongo shell, we'll use the collection's function ***aggregate***.

Let's assume we would like a query that returns the number of receipts.

In Mongo Shell, run the following query with the query pipeline

JavaScript

```javascript
db.salesReceipts.aggregate(
    [
        {
            $count: "number_of_receipts"
        }
    ]
)
```

Let's expand on this example slightly. Let's count all the sales receipts where "Milk" was sold.

JavaScript

```javascript
db.salesReceipts.aggregate(
    [
        {
            $match: {"SalesDetail.ItemName":"Milk"}
        },
        {
            $count: "number_of_receipts"
        }
    ]
)
```

Finally let's return the receipt number and the total of each receipt.

JavaScript

```javascript
db.salesReceipts.aggregate(
    [
        {
            $unwind: "$SalesDetail"
        },
        {
            $group:
```

```
            {
                _id: "$ReceiptNo"
                , ReceiptSales: { $sum: { $multiply: ["$SalesDetail.quanti-
ty", "$SalesDetail.priceperunit"]}}
            }
        },
        {
            $project:
            {
                ReceiptNumber: "$_id"
                , ReceiptSales: {$round: ["$ReceiptSales",2]  }
                , _id:  0 // projecting with value zero will not display
the column named _id
            }
        }
    ]
)
```

> **Note**
>
> You can also run these *db.collection.aggregate* functions in the Azure portal by using the *Shell* included in the Azure Cosmos DB for MongoDB account page. Under *Data Explorer*, expand your database and choose the collection you want to run the function against, you will see a *New Shell* option in the collection menu.

As we've illustrated, Azure Cosmos DB for MongoDB uses MongoDB aggregation pipelines to give us the ability of running complex queries right at the server with no issues. This reduces the need to send all the original documents back to the client to calculate those aggregations programmatically.

Next unit: Exercise – Indexes and aggregation pipelines

Continue >