Data Transformation DSL

Problem

- Big data transformation
- Possibly complicated calculations and algorithms

Requirements for a Solution

- scalable
- maintainable
- understandable by non technical person

Approach

- Avoid random access
- Process everything sequentially
- “Sequentially functional”

Why DSL?

- Simple usable interface
- Pluggable implementations
- Declarative approach

Data

- Database is defined as one or more collections
- Collection is an array of Tuples (Rows)
- Tuple has one or more untyped fields

How does it work?

- Chain/Graph of operations
- Similar to Unix pipe just not only linear
- Expression evaluation
- Potential for high degree of parallelization

Basic operations

- define_collection
- generate
- project
- filter
- aggregate
- compose
- group
- sort

define_collection collection_name, *fields

Loads a collection (e.g. from a csv file) generate new_collection, field, count, &block

Generates a new collection with count rows containing one field. The block evaluation determines the value in each row.

project new_collection, collection, options

Can add or remove fields from a collection. options is map specifying which fields should be included or excluded. It can also contain lambdas to calculate new field values.

shortcut:
calculate new_collection, collection, field, &block

Creates a new collection with just one field per row and value calculated in block.

filter new_collection, collection, &block

Retains rows for which block evaluates to true

aggregate new_collection, collection,
initial_value, field, &block

Performs an aggregation over all rows of a collection returning a new collection with single row and single field compose new_collection, *collections, &block

It's the classical join, but default is not cartesian product.

group new_collection, collection, options

Groups records according to specification in options.

options[:fields] - fields that drive the group by

options[:computations] - mapping between new fields and lambdas to calculate those fields

Sample Implementation

- Calculate impact of discount campaigns
- Leverages exponential smoothing

What is exponential Smoothing?

- popular schema to produce a smoothed Time series
- Single Moving Average - observations weighted equally
- Exponential Smoothing - older observations get exponentially decreasing weights.

Exponential Smoothing

$$s_1 = x_0$$
$$s_t = \alpha x_{t-1} + (1-\alpha) s_{t-1}, t > 1$$

α is the *smoothing factor*, and 0 <= α <= 1.

The time series look like geometric progression

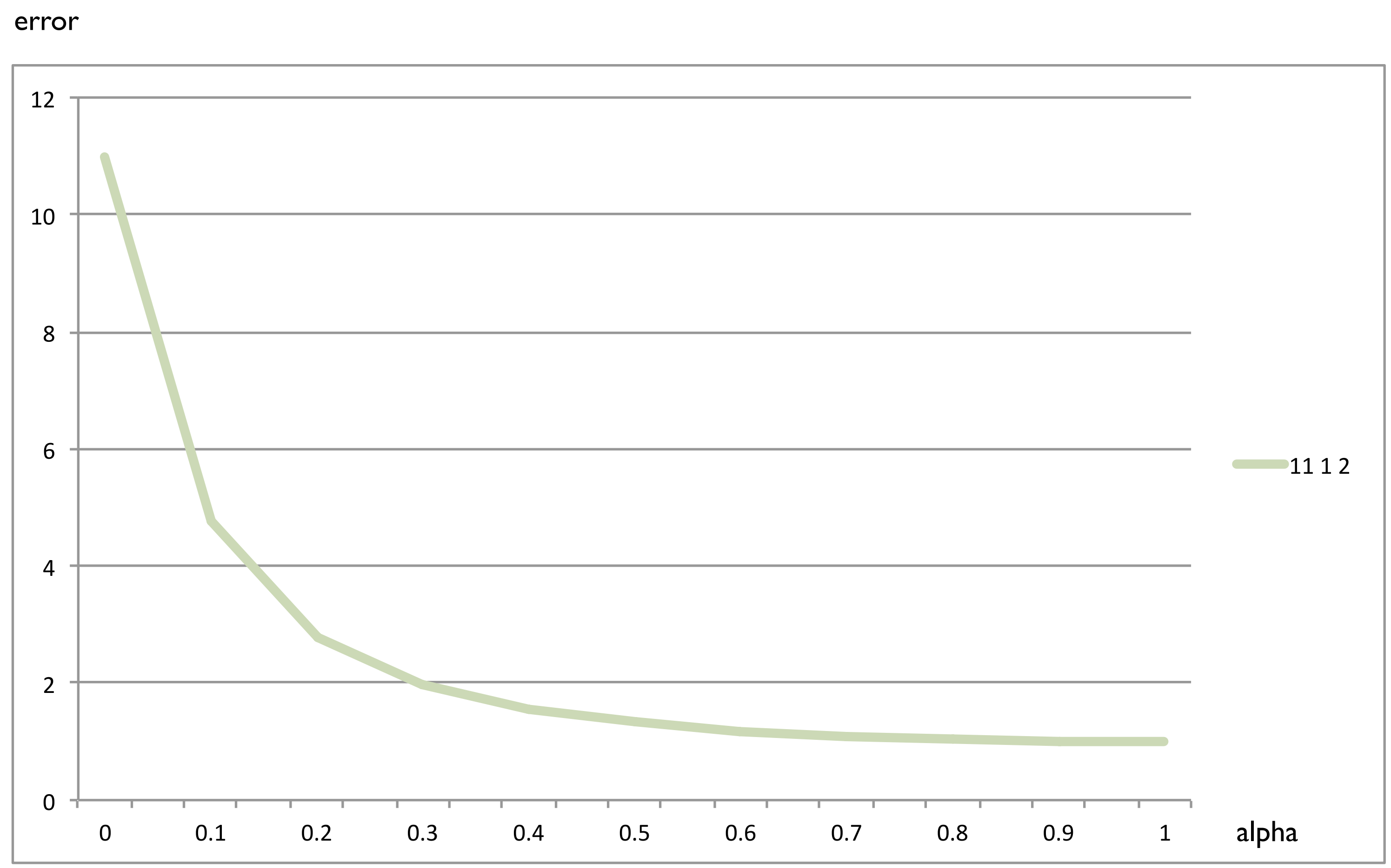
$$\begin{aligned} s_t &= \alpha x_{t-1} + (1-\alpha) s_{t-1} \\ &= \alpha x_{t-1} + \alpha(1-\alpha) x_{t-2} + (1-\alpha)^2 s_{t-2} \\ &= \alpha \left[x_{t-1} + (1-\alpha) x_{t-2} + (1-\alpha)^2 x_{t-3} + (1-\alpha)^3 x_{t-4} + \cdots \right] + (1-\alpha)^{t-1} x_0. \end{aligned}$$

Optimal Alpha

- sum of the quantities $(sn_{-1} - xn_{-1})^2$ is minimized
- optimal alpha is only valid for a particular sequence
- no correlation to entire data variance

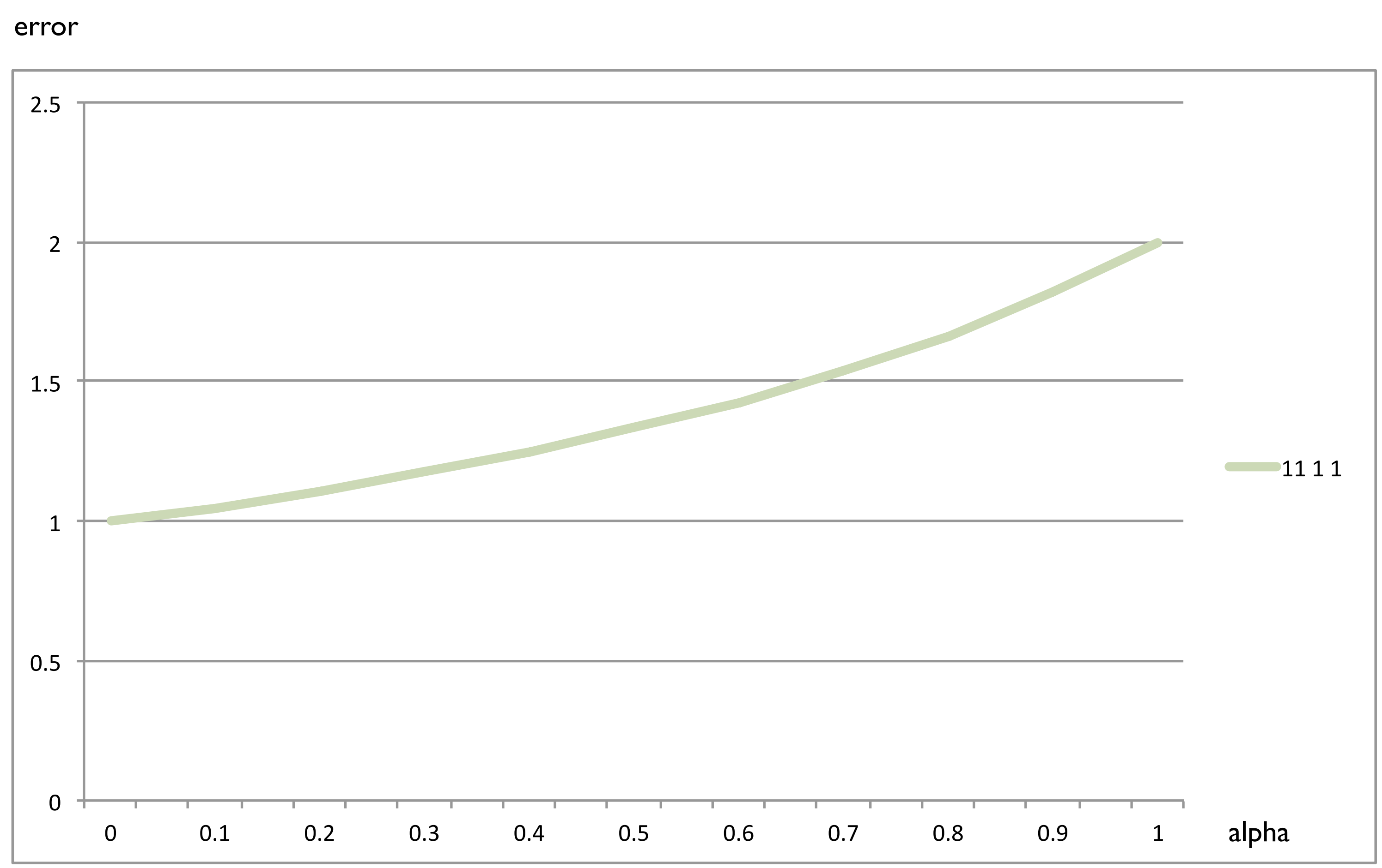
Example

2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1



Example

1,2,1,1,1,1,1,1,1,1,1,1



Additional problem

- Linear stretching

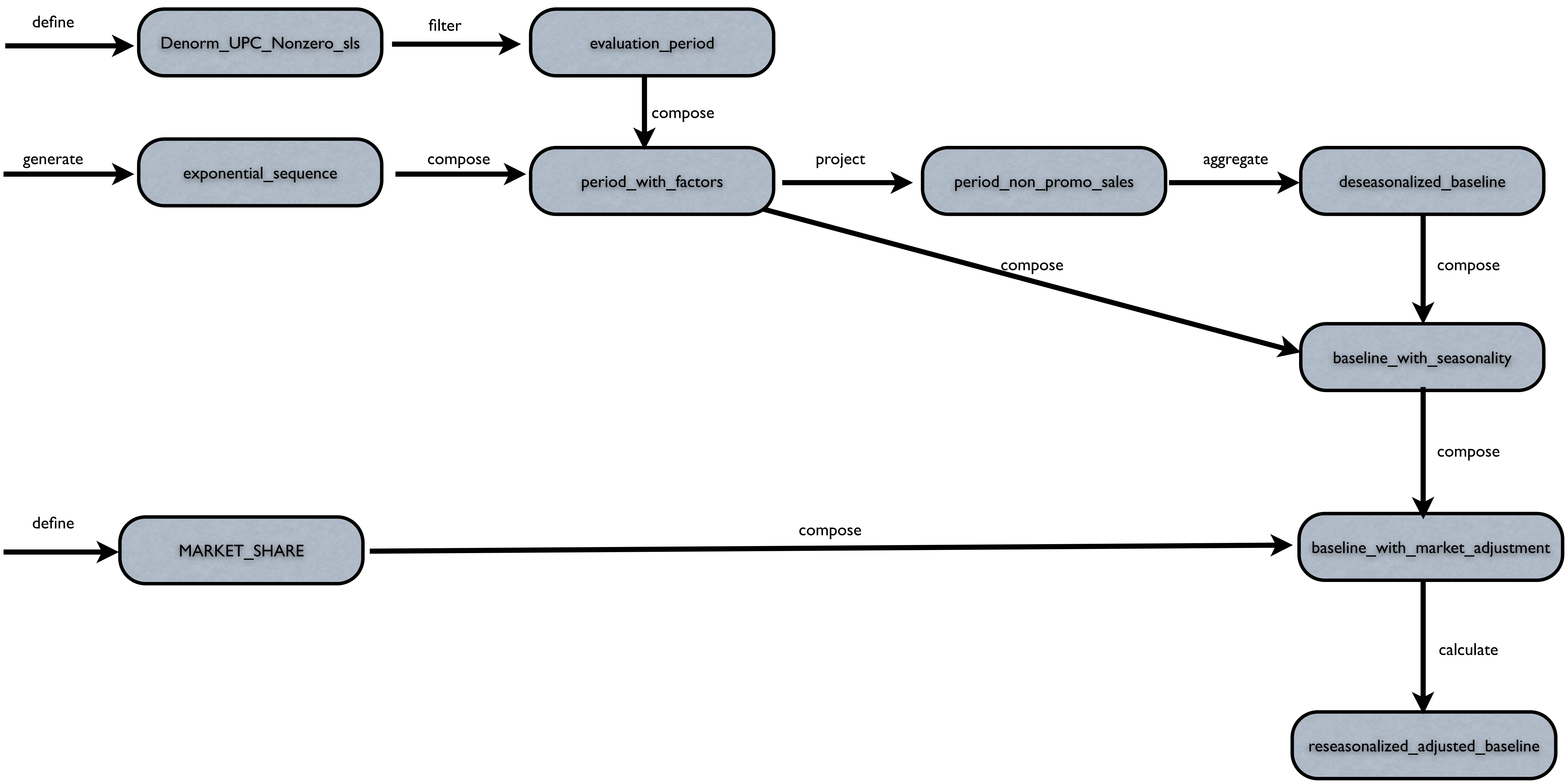
Implementation

- Seemed at first impossible without random access
- Multiple iterations
- How do you do loops in declarative language?
- etc.

The code

```
Transform::Dsl.draw do
  define_collection :Denorm_UPC_Nonzero_sls,
                    :UPC_NBR, :WEEK_ID, :FS_SUB_CATEGORY_ID, :FS_SALES_DLRS, :FS_COSTS_DLRS, :FS_PROMO_SALES_DLRS,
                    :FS_VENDOR_FUNDING_DLRS, :FS_SALES_UNITS, :FS_PROMO_SALES_UNITS, :FS_NBR_TRANSACTIONS,
                    :FS_AVG_BASKET_SIZE, :FS_UNADJUSTED_MARGIN_DLRS, :FS_Lifecycle_stage, :DP_PROMO_FLAG,
                    :DC_WEEK_NUM, :DC_YEAR_NUM, :DC_WEEK_DATE, :DC_AD_WEEK, :DS_SUB_CATEGORY_ID, :DS_SEASONALITY_INDEX,
                    :FN_start_sale_week, :FN_end_sale_week, :ADJ_SALES_DLRS
  define_collection :MARKET_SHARE, :FS_SUB_CATEGORY_ID, :ADJUSTMENT_FACTOR
  filter(:evaluation_period, :Denorm_UPC_Nonzero_sls) do |sale|
    sale.UPC_NBR.to_i == 28 && (sale.DC_WEEK_NUM.to_i-20).abs <= 8
  end
  generate(:exponential_sequence, :factor, 17) { |i| i==8 ? 0 : (0.5**((i - 8).abs+1))/(1-0.5**8) }
  compose(:period_with_factors, :evaluation_period, :exponential_sequence)
  project(:period_non_promo_sales, :period_with_factors,
          pwk_sales: lambda { |week| week.DP_PROMO_FLAG.to_i == 1 ? week.previous.pwk_sales : week.FS_SALES_DLRS })
  aggregate(:deseasonalized_baseline, :period_non_promo_sales, 0, :deseasonalized_baseline) do |total, week|
    total + week.pwk_sales.to_f*week.factor.to_f/week.DS_SEASONALITY_INDEX.to_f
  end
  compose(:baseline_with_seasonality, :deseasonalized_baseline, :period_with_factors) do |baseline, week|
    week.factor == "0"
  end
  compose(:baseline_with_market_adjustment, :baseline_with_seasonality, :MARKET_SHARE) do |baseline, market|
    baseline.FS_SUB_CATEGORY_ID == market.FS_SUB_CATEGORY_ID
  end
  calculate(:reseasonalized_adjusted_baseline, :baseline_with_market_adjustment, :reseasonalized_adjusted_baseline) do |week|
    week.deseasonalized_baseline.to_f * week.DS_SEASONALITY_INDEX.to_f * week.ADJUSTMENT_FACTOR.to_f
  end
  store :reseasonalized_adjusted_baseline
end
```

Graphical representation



```
define_collection :Denorm_UPC_Nonzero_sls,
                  :UPC_NBR, :WEEK_ID, :FS_SUB_CATEGORY_ID, :FS_SALES_DLRS, :FS_COSTS_DLRS, :FS_PROMO_SALES_DLRS,
                  :FS_VENDOR_FUNDING_DLRS, :FS_SALES_UNITS, :FS_PROMO_SALES_UNITS, :FS_NBR_TRANSACTIONS,
                  :FS_AVG_BASKET_SIZE, :FS_UNADJUSTED_MARGIN_DLRS, :FS_Lifecycle_stage, :DP_PROMO_FLAG,
                  :DC_WEEK_NUM, :DC_YEAR_NUM, :DC_WEEK_DATE, :DC_AD_WEEK, :DS_SUB_CATEGORY_ID, :DS_SEASONALITY_INDEX,
                  :FN_start_sale_week, :FN_end_sale_week, :ADJ_SALES_DLRS
```

```ruby
filter(:evaluation_period, :Denorm_UPC_Nonzero_sls) do |sale|
  sale.UPC_NBR.to_i == 28 && (sale.DC_WEEK_NUM.to_i-20).abs <= 8
end
```

```
generate(:exponential_sequence, :factor, 17) { |i| i==8 ? 0 : (0.5**((i - 8).abs+1))/(1-0.5**8) }
```

```
project(:period_non_promo_sales, :period_with_factors,
        pwk_sales: lambda { |week| week.DP_PROMO_FLAG.to_i == 1 ? week.previous.pwk_sales : week.FS_SALES_DLRS })
```

```ruby
aggregate(:deseasonalized_baseline, :period_non_promo_sales, 0, :deseasonalized_baseline) do |total, week|
  total + week.pwk_sales.to_f*week.factor.to_f/week.DS_SEASONALITY_INDEX.to_f
end
```

```
compose(:baseline_with_seasonality, :deseasonalized_baseline, :period_with_factors) do |baseline, week|
  week.factor == "0"
end
```

```
calculate(:reseasonalized_adjusted_baseline, :baseline_with_market_adjustment, :reseasonalized_adjusted_baseline) do |week|
  week.deseasonalized_baseline.to_f * week.DS_SEASONALITY_INDEX.to_f * week.ADJUSTMENT_FACTOR.to_f
end
```

```
store :reseasonalized_adjusted_baseline
.
```

Demo

Problems

- sort
- compose (join)
- group

Goal

- computational complexity O(data size)
- current solution: 10,000 rows/sec

We are not the only one

MongoDB Aggregation Framework

New framework currently in 2.2.0-rc1

- `$project`
- `$match`
- `$limit`
- `$skip`
- `$unwind`
- `$group`
- `$sort`

Anyone missing an operation?

Anyone missing an operation?

join

Example Script

```
db.article.aggregate
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : “$tags” },
  { $group : {
     _id : { tags : 1 },
     authors : { $addToSet : “$author” }
  } }
);
```

Differences

- Only sequential pipe, no graph
- requires de-normalization of data into MongoDB

Questions

- Will users want to use this DSL?
- Is it easy to express problems with it?
- Will we be able to express all cases?
- Can it scale without user paying attention to how the problem is specified?

References

- http://github.intranet.mckinsey.com/Heinrich-Klobuczek/transform
- http://docs.mongodb.org/manual/applications/aggregation/