Technical Guide

Contents

Glossary 3

Solution Overview 3

Architecture 5

Structure of Configuration File 5

Switching from Demo to Production Mode 7

Power BI Visualization Guide 8

  Dashboard Content 8

  Configuration 8

    Download the Power BI report file and sign-in 8

    Update the Azure Data Lake Store account in the Power BI file 9

    Select the active policy being compared with the baseline policy 10

    [Optional] Publish the dashboard to Power BI online 11

Website Guide 13

Structure of Directories in Data Lake 14

How to Integrate New Solvers 15

How to Define New Inventory Management Policies 18

  Define Abstract Optimization Problem using Pyomo 18

  Write USQL Script that Computes Known Values of Optimization Problem 19

  Write USQL Script that Places Orders 20

  Update Configuration File 21

Data Schema 21

  Static Datasets 21

    Store 21

    Store Storage 22

    Store Departments 22

    Brands 22

    Brands Products 22

    Store Department Brand Products 23

    Store Product Storage 23

    Suppliers 24

Store Product Supplier 24
Dynamic Datasets 25
Demand Forecast 25
Sales 25
Inventory Levels 26
Orders 26
Integration with External Sources 27
Structure of Simulator 27
Simulation of predicted and actual demand 28
Sample Inventory Management Policies 29
(s, Q) policy 29
Definition 29
Optimization problem for generating the policy 30
Implementation of optimization problem in Pyomo 30
(s, Q) policy for perishable items 31
Definition 31
Optimization problem for generating the policy 31
Implementation of optimization problem in Pyomo 32
(R, S) policy 33
Definition 33
Optimization problem for generating the policy 33
Implementation of optimization problem in Pyomo 33
EOQ policy 34
Definition 34
Optimization problem for generating the policy 35
Implementation of optimization problem in Pyomo 35
EOQ_big: Large-scale implementation of EOQ policy 35
Dynamic Lot Sizing policy 36
Definition 36
Optimization problem for generating the policy 36
Implementation of optimization problem in Pyomo 37
Capacitated Dynamic Lot Sizing policy 37
Definition 37

Optimization problem for generating the policy 38
Implementation of optimization problem in Pyomo 38
Newsvendor policy 39
Definition 39
Optimization problem for generating the policy 39
Implementation 40

Glossary

Inventory optimization – Determine the optimal timing and size of orders to optimize certain business metrics, e.g. maximize turn-over rate.

Inventory management policy - A rule that specifies when to make order and how many items to order. Inventory management policy is found by solving optimization problem. For example, in sQ policy, if the number of items in inventory is *s*, then order additional *Q* items. Optimal values of *s* and *Q* are found by solving optimization problem and can change over time.

Inventory level - Number of products in inventory.

Solution Overview

In this Solution How-To Guide, we develop a cloud-based, scalable and flexible inventory optimization solution for retail. To scale up to hundreds of millions of store and product combinations, we use Azure Data Lake Analytics (ADLA) for data processing and Azure Batch for solving optimization problems in parallel.

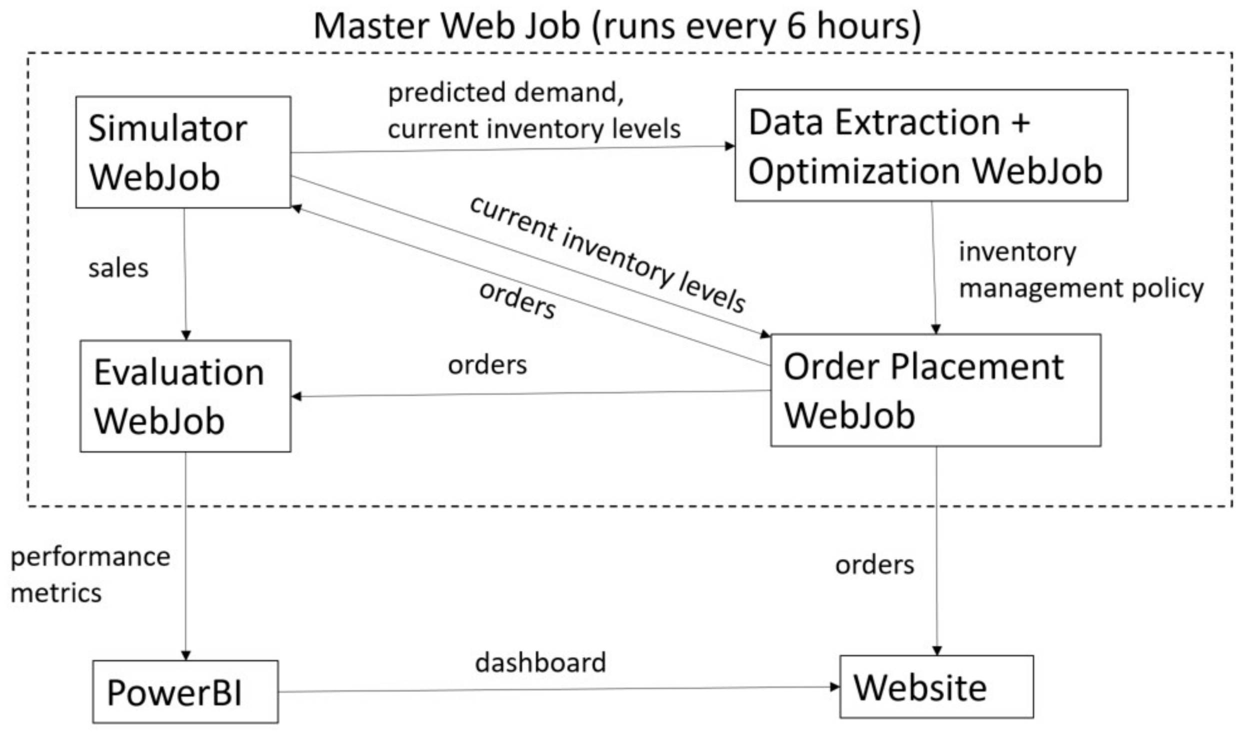
We use Pyomo library to formulate optimization problems and also as an interface with optimization solvers. We provide scripts for eight commonly used inventory optimization policies. These scripts can be customized for a specific retailer and new policies can be added by providing a few scripts. For details of inventory policies included, see the section **Sample Inventory Management Policies**. To add a new policy, see the section **How to Define New Inventory Management Policies.**

For solving optimization problems, we included Bonmin, an open-source solver for general MINLP (Mixed Integer NonLinear Programming) problems, in a Docker image. This Docker image is used by virtual machines in Azure Batch pool to solver thousands of optimization problems in parallel. Additional open-source solvers and commercial solvers like MIPCL and Gurobi can be easily incorporated into this Docker image, see the section on **How to Integrate New Solvers.**

The inventory optimization solution is composed of five major components: data simulation, data pre-processing, inventory optimization, order placement, and evaluation. The table below lists the function and execution engine of each component. Each step starts with an Azure Web Job. The actual execution is done by Python scripts within the web jobs, Azure Data Lake Analytics jobs submitted by the web job Python script, or Azure Batch jobs submitted by the web job Python script.

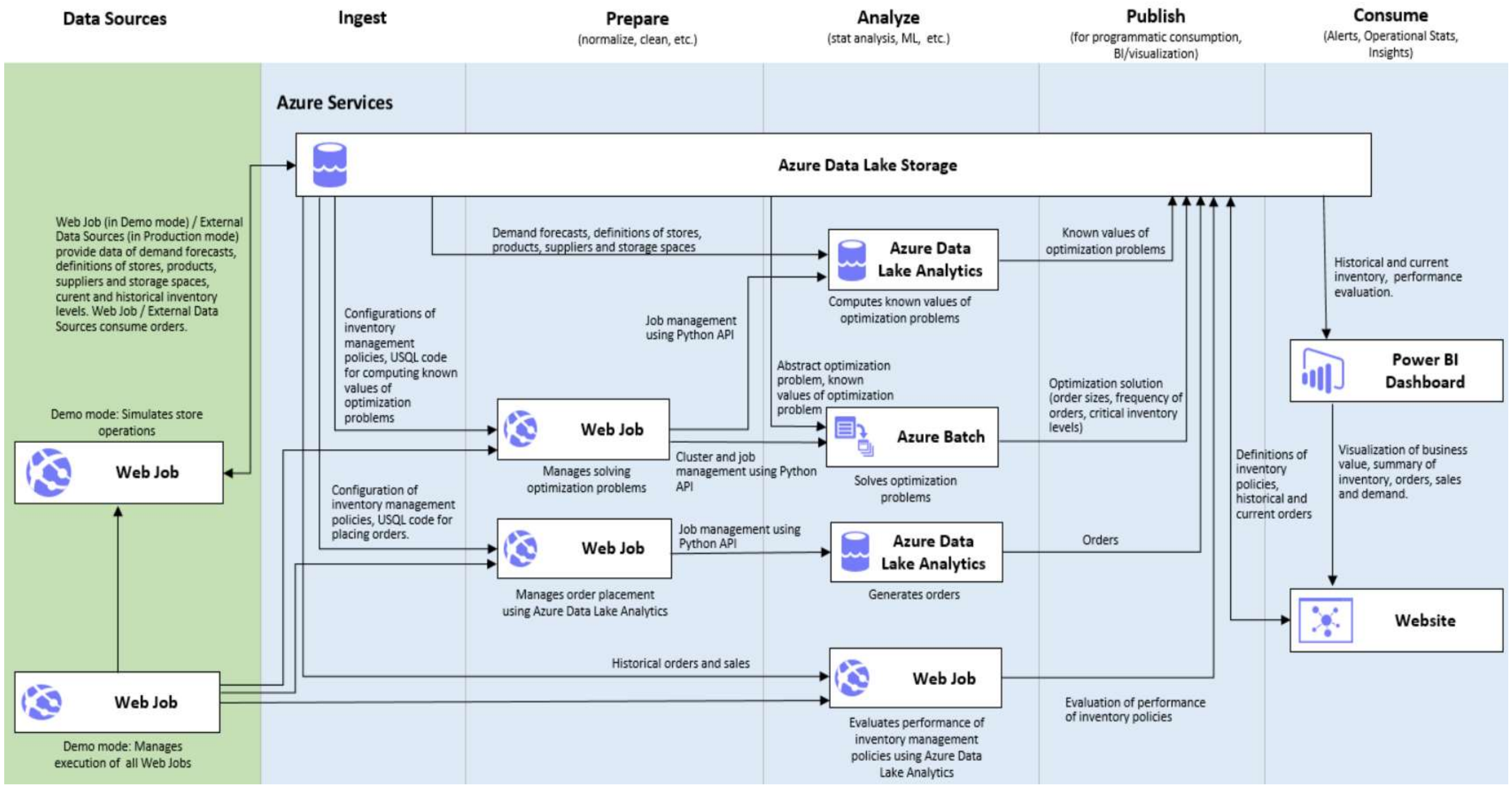
Web Job Name	**Function**	**Execution Engine**
Simulator	Simulate raw data: stores, storage spaces, products, suppliers, demand forecasting, sales, and inventory levels, order placements and deliveries.	Python script in the web job
InventoryOptimization (Data extraction part)	Use Pyomo to define abstract optimization problems of inventory management policies. Extract known values of optimization problems from the raw data.	Pyomo Python scripts in the web job. U-SQL jobs on Azure Data Lake Analytics
InventoryOptimization (Optimization part)	Create inventory management policy by solving inventory optimization problems using Bonmin.	Azure Batch
GenerateOrder	Use solution optimization problem, current time and inventory levels to place new orders.	U-SQL jobs on Azure Data Lake Analytics
Evaluation	Compute performance metrics of inventory management policies.	Python script in the web job

For demo purpose, a master web job is scheduled to run every 6 hours and invokes the other web jobs to simulate one day every 6 hours. The figure below illustrates the data flow between different web jobs. Note that all web jobs write/read to/from Azure Data Lake Store (ADLS). For example, the Simulator web job first writes simulated sales data to ADLS and the evaluation web job reads the sales data from ADLS. To run the solution in production mode, see the section on **Switching from Demo to Production Mode.**



We also provide a PowerBI dashboard and a website to visualize and utilize the optimization results. See the sections **PowerBI Visualization Guide** and **Website Guide** for more details.

Architecture


Data Sources
Ingest
Prepare
(normalize, clean, etc.)
Analyze
(stat analysis, ML, etc.)
Publish
(for programmatic consumption, BI/visualization)
Consume
(Alerts, Operational Stats, Insights)
Azure Services
Azure Data Lake Storage
Web Job (in Demo mode) / External Data Sources (in Production mode) provide data of demand forecasts, definitions of stores, products, suppliers and storage spaces, curent and historical inventory levels. Web Job / External Data Sources consume orders.
Demand forecasts, definitions of stores, products, suppliers and storage spaces
Azure Data Lake Analytics
Computes known values of optimization problems
Known values of optimization problems
Historical and current inventory, performance evaluation.
Configurations of inventory management policies, USQL code for computing known values of optimization problems
Job management using Python API
Abstract optimization problem, known values of optimization problem
Optimization solution (order sizes, frequency of orders, critical inventory levels)
Power BI Dashboard
Demo mode: Simulates store operations
Web Job
Web Job
Manages solving optimization problems
Cluster and job management using Python API
Azure Batch
Solves optimization problems
Definitions of inventory policies, historical and current orders
Visualization of business value, summary of inventory, orders, sales and demand.
Configuration of inventory management policies, USQL code for placing orders.
Web Job
Manages order placement using Azure Data Lake Analytics
Job management using Python API
Azure Data Lake Analytics
Generates orders
Orders
Website
Historical orders and sales
Web Job
Evaluates performance of inventory management policies using Azure Data Lake Analytics
Evaluation of performance of inventory policies
Web Job
Demo mode: Manages execution of all Web Jobs

Structure of Configuration File

The **Configurations.xlsx** file is used to configure which inventory policies are active, how frequent each policy needs to be run, which solver should be used to solve the optimization problem, etc. This file is located in the **configuration** folder in Azure Data Lake Store. We explain the fields in each sheet of this file in the following paragraphs.

InventoryPolicyConfig

- **ActiveFlag:** 1 – Active, 0 – Inactive. If active, the inventory policy will be checked if any computation needs to be done every time the web jobs InventoryOptimization and GenerateOrder are invoked. If inactive, the policy will not be checked.
- **InventoryPolicyID:** A unique ID identifying the inventory policy.

- **ScheduleID_Optimization:** When joined with the **ScheduleID** field in the **ScheduleConfig** sheet, determines the schedule of the optimization tasks for each policy. If the policy is active, the web job will check if any optimization task needs to be executed for the current period of time. For example, the InventoryOptimization web job can be scheduled to run once every day. For policies that are scheduled to run once every day, optimization tasks will be executed every time the InventoryOptimization web job runs. For policies that are scheduled to run once every week, optimization tasks will only be executed one out of the seven times the InventoryOptimization web job run every week.
- **ScheduleID_GenerateOrder:** When joined with the **ScheduleID** field in the **ScheduleConfig** sheet, determines the schedule of order generation for each policy. The schedule mechanism of GenerateOrder web job is the same as that of the InventoryOptimization web job.
- **InventoryPolicyName:** The name of the inventory policy.
- **SolverName:** The solver used to solve the optimization problem for each policy. This field should be joined with the **SolverName** field in the **SolverConfig** sheet to get the absolute path and input file format of the solver.
- **OptimizationDefinition:** The name of the Pyomo python script that defines the optimization problem, without the .py extension. For example, when the value of the field is "sQ", an **sQ**.py file should be located in the /inventory_scripts/sQ/ in Azure Data Lake Store.
- **PartitionFields:** How the data are partitioned in Azure Data Lake Store. This field also determines the granularity of the optimization problem to solve. For example, when the value of this field is "Store,Product", one optimization problem is solved for each (Store,Product) combination.
- **DirectoryName:** Sub-directory of scripts under the inventory_scripts folder in Azure Data Lake store. For example, when the value of this field is sQ, all the scripts of the sQ policy should be located in the folder /inventory_scripts/sQ/.
- **USQLCreateCSV:** Names of the U-SQL scripts that convert raw data to Pyomo input format. These scripts are located in the /inventory_scripts/<DirectoryName>/ folder in Azure Data Lake Store.
- **USQLOrder:** Names of the U-SQL scripts that convert optimization results to order. These scripts are located in the /inventory_scripts/<DirectoryName>/ folder in Azure Data Lake Store.
- **USQLTimeout:** The Azure Data Lake Analytics jobs will be cancelled if they don't complete within this timeout limit. The unit is minute. When multiple policies are running, all the timeout limits will be summed up.
- **OptimizationTimeout:** The Azure Batch jobs will be cancelled if they don't complete within this timeout limit. The unit is minute. When multiple policies are running, all the timeout limits will be summed up.

ScheduleConfig

- **ScheduleID:** A unique ID identifying each schedule definition.
- **CronExpression:** The cron expression of the schedule.
- **StartDate:** The start date of the schedule.
- **EndDate:** The end date of the schedule.
- **ScheduleDescription:** A description of the schedule.

SolverConfig

- **SolverName:** A unique name of the solver.
- **SolverPath:** The path of the solver in the Docker image. See the section **How to Integrate New Solvers** for more details.
- **DefinitionFileExtension:** The file extension of the optimization problem definition file, i.e. input file to the solver.

Switching from Demo to Production Mode

For demonstration purpose, this solution is set up in a demo mode. In this demo mode, the **Main** web job is scheduled to run once every 6 hours and uses a simulation date time stored in the file **/webjob_log/LastSimulationDatetime.txt** in Azure Data Lake Store to keep track of the simulation date time. The **Main** web job increases the date time in this file by one day every 6 hours. All the other web jobs take an optional argument -d or --datetime. If this argument is given, the web jobs uses the value of this argument as the current date time. Therefore, we simulate one day every 6 hours in the demo mode.

When the -d or --datetime argument is not given, the web jobs uses the actual date and time. This is called the production mode. In order to switch to production mode, you need to delete/disable the **Main** web job and upload a settings.job file with desired schedule to each web job.

First, you can get a template of the settings.job file from the **Main** web job. To do this

- Navigate to portal.azure.com
- On the left tab click **Resource Groups**
- Click on the resource group of the deployed solution
- Click the **App Service** in the list of resources.
- Under **App Service**, search for **Advanced Tools** and select it. Click **Go** in the new blade and you will be taken to the Kudu console.
- At the top of the Kudu console, click **Debug console -> CMD.**
- Navigate to the folder **site/wwwroot/app_data/jobs/triggered**. This folder contains all the scripts of the web jobs.
- Click on the **Main** folder. The settings.job file contains the schedule (in cron expression) of the **Main** web job. Click the download button on the left side of the file and save it to your computer.

Now, you can delete the **Main** web job.

- Return to the **App Service** page
- Search for **Webjobs** and select it
- Select the **Main** web job and click **Delete** at the top

Last, create a settings.job file with desired schedule for each web job and upload it to the web job folder in the Kudu console. This will enable each web job run independently on its own schedule.

Alternatively, if you want to keep the **Main** web job calling the other web jobs, you will need to modify the Python script **masterwebjob.py** in the **Main** web job folder, so that it doesn't pass the simulated date time argument to the other web jobs. You can edit the file in the Kudu console by clicking the pencil icon on the left side of the file. You can also download the file and edit it on your computer. After editing, you can drag and drop the file to the Kudu console folder.

Power BI Visualization Guide

In this section, we introduce the information contained in the Power BI dashboard and how to configure the dashboard.

Dashboard Content

The dashboard contains four parts:

1. **Inventory Management Policies**: shows the inventory management policies that have been created in a table named *Inventory Policy List*, where Active Flag indicates which policy is active now. Currently, we have two active policies namely policies with Policy ID *s_Q* and *s_Q_perishable*. Besides, there is a policy named *Sim* used as a baseline for comparison purposes.

2. **Performance Evaluation:** presents four evaluation metrics of the inventory optimization, namely Normalized Revenue (NR), Total Revenue (TR), Number of Stockout Events (NSE), and Turnover Ratio (TOR). These metrics are often used to evaluate the performance of a certain inventory optimization policy from different perspectives. Their definitions are briefly explained below

    - TR is the total amount of sales during a given period.
    - NR is the total revenue divided by the number of days during which products generating the revenue have been in the inventory.
    - NSE is the estimated number of products that are out of stock.
    - TOR is the total revenue divided by the average inventory.

    For each metric, we visualize the value averaged across a certain set of stores, the value averaged over both all the stores and a recent period (last week, last month, last quarter), as well as the improvement of a certain active policy over the baseline policy.

3. **Inventory Status**: displays the inventory level aggregated over all the products of a specific set of stores at the end of each day.

4. **Store Information**: shows the information of the stores which have been simulated.

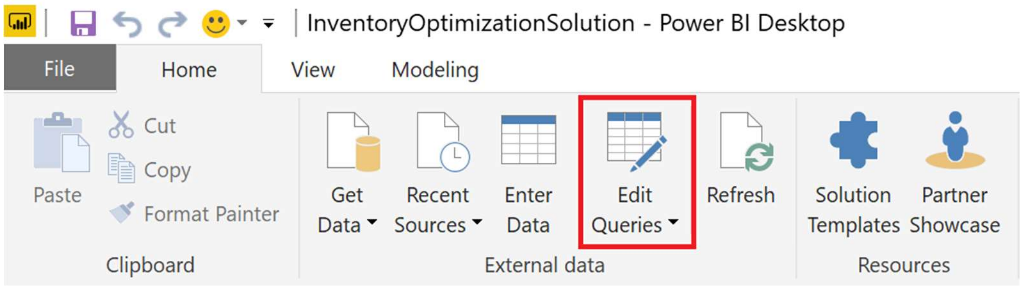
Configuration

Download the Power BI report file and sign-in

- Make sure you have installed the latest version of Power BI desktop. If not, you may see an error message showing that it is unable to open the dashboard when you open the Power BI file.
- In this GitHub repository, you can download the Power BI file of the dashboard *InventoryOptimizationSolution.pbix* under the folder Power BI and then open it by double clicking.
- The dashboard is prepopulated with cached data so that you could have a quick glance at the results of the solution. You might see a message saying "There are pending changes in your queries that haven't been applied." Please **DO NOT** Apply Changes since the data sources have not been updated yet.

Note: The visualizations that you see now in your Power BI report are the cached results from a previous demo deployment, rather than the real data in your own deployment. You will see the visualizations that correspond to the data sitting under your subscription after you follow the below steps to change the data connection to your own Azure Data Lake Store.

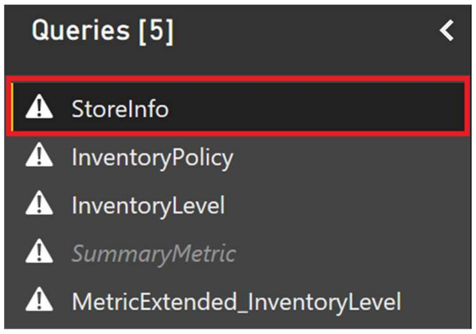
- Sign in by clicking **Sign in** on the top-left corner. Note: You must have a Microsoft Office 365 subscription for Power BI access.
- Click **Edit Queries** on the top and open the query editor. You will see 5 Queries in the left panel of the query editor. You might also see an error message saying "DataFormat.Error: Invalid URI: The hostname could not be parsed.". Please ignore this error message for now and follow the instructions below to update the data sources. Once the data sources are updated, the error will be gone.



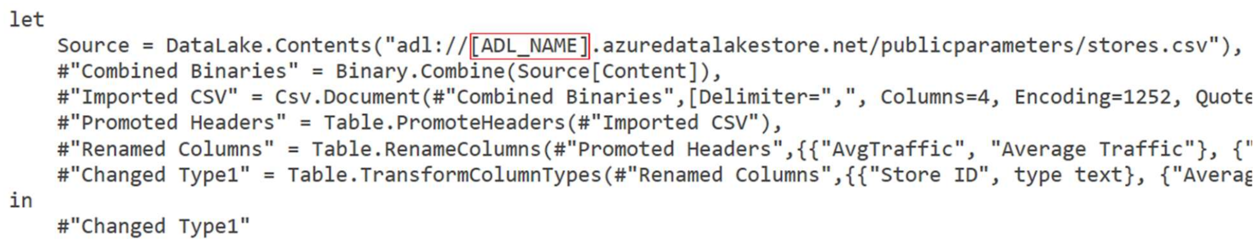
Update the Azure Data Lake Store account in the Power BI file

Next, you will need to take the following steps to update the data sources:

- Click **StoreInfo** query and you will see that this query is highlighted in a darker color as the following screenshot shows. Then, click the **Advanced Editor** on the top, which is next to the **Refresh Preview**.

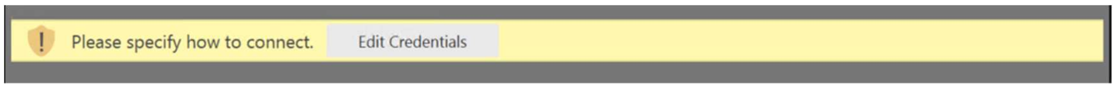


- On the pop-up Advanced Editor window, replace the **[ADL_NAME]** in the first line with the name of the Azure Data Lake Store that you deployed in the previous steps.

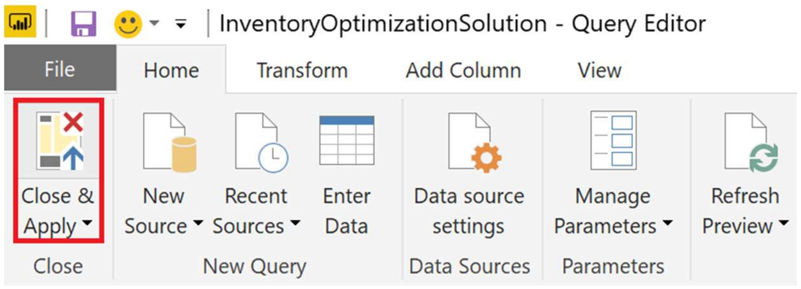


```
let
    Source = DataLake.Contents("adl://[ADL_NAME].azuredatalakestore.net/publicparameters/stores.csv"),
    #"Combined Binaries" = Binary.Combine(Source[Content]),
    #"Imported CSV" = Csv.Document(#"Combined Binaries",[Delimiter=",", Columns=4, Encoding=1252, Quote
    #"Promoted Headers" = Table.PromoteHeaders(#"Imported CSV"),
    #"Renamed Columns" = Table.RenameColumns(#"Promoted Headers",{{"AvgTraffic", "Average Traffic"}, {"
    #"Changed Type1" = Table.TransformColumnTypes(#"Renamed Columns",{{"Store ID", type text}, {"Averag
in
    #"Changed Type1"
```

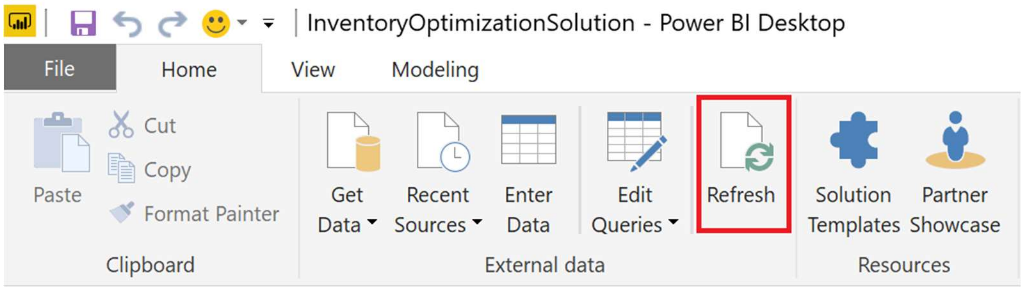
- Then, click **Done** on the bottom-right corner of the Advanced Editor window. You will see a warning message below


Please specify how to connect.

- Click **Edit Credentials** in the above message and sign in the Azure Data Lake Store that you deployed by clicking **Sign in** in the pop-up window with your Azure account name and password.
- Repeat the above steps that you have done for **StoreInfo** query (click the corresponding query, open the corresponding advanced query editor and replace the **[ADL_NAME]** with your data lake store name) on the other 4 queries: **InventoryPolicy** query, **InventoryLevel** query, **MetricExtended_InventoryLevel** query, and **SummaryMetric** query.
- Click **Close & Apply** on the top-left, and you will see the visualization report in Power BI Desktop.



- [Optional] You can click **Refresh** on the top if you want to refresh the report, when there are new data coming in.

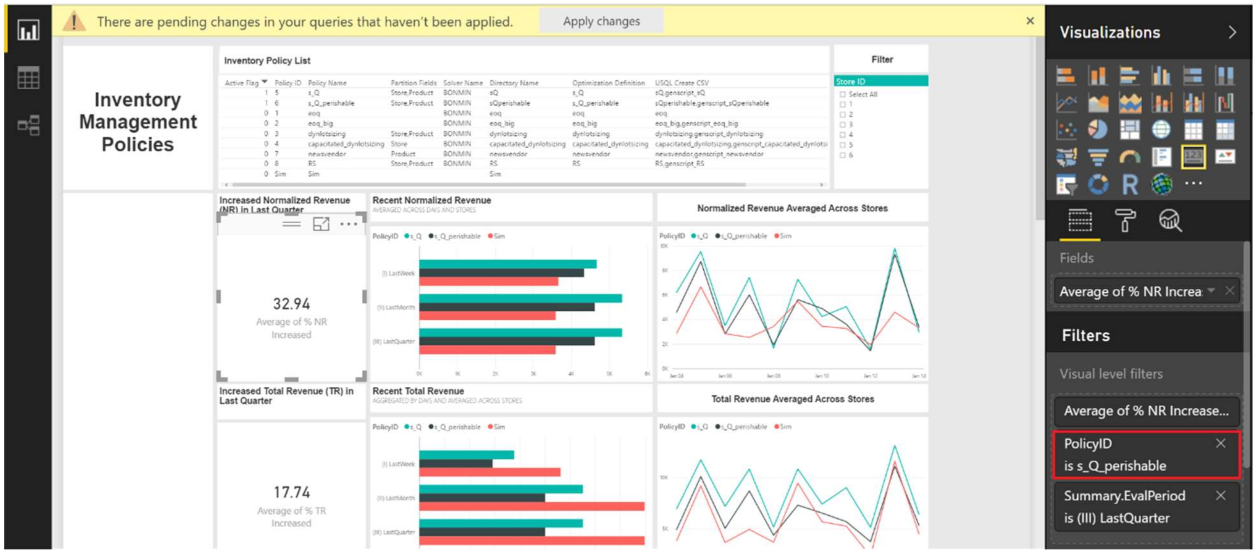


Select the active policy being compared with the baseline policy

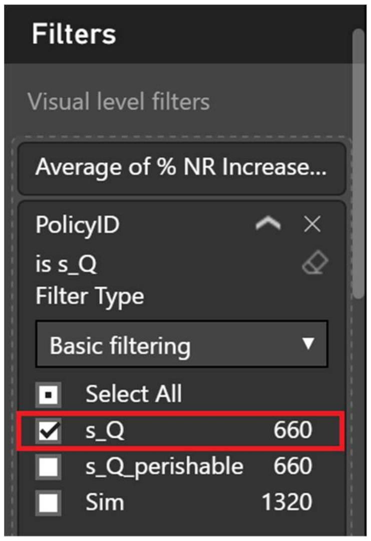
The dashboard shows the performance metrics of two active inventory optimization policies and a baseline policy. The policy IDs of the active policies are s_Q and s_Q_perishable respectively, while the policy ID of the baseline policy is Sim. By comparing an active policy with the baseline policy, we will see the performance improvements generated by inventory optimization.

Note: In the dashboard, the improvements of s_Q_perishable policy over the baseline policy are displayed *by default*. The current step is needed if you want to change the active policy being compared with the baseline policy to s_Q.

We can choose which active policy to be compared with the baseline policy when computing the improvement ratio of each metric shown in each of the 4 cards in the Performance Evaluation part of the dashboard. For instance, you can click the card indicating the increased normalized revenue in the last quarter as follows


There are pending changes in your queries that haven't been applied.
Apply changes
Inventory Management Policies
Inventory Policy List
Filter
Increased Normalized Revenue (NR) in Last Quarter
32.94
Average of % NR Increased
Recent Normalized Revenue
Normalized Revenue Averaged Across Stores
Increased Total Revenue (TR) in Last Quarter
17.74
Average of % TR Increased
Recent Total Revenue
Total Revenue Averaged Across Stores
Visualizations
Fields
Average of % NR Increa
Filters
Visual level filters
Average of % NR Increase...
PolicyID is s_Q_perishable
Summary.EvalPeriod is (III) LastQuarter

On the right-hand side, you can find that **PolicyID is s_Q_perishable** under the **Visual level filters** of the **Filters** panel. If you move your cursor to **PolicyID is s_Q_perishable**, you will see an arrow sign appearing. You can further click this arrow sign and choose s_Q policy instead of s_Q_perishable policy. Typically you should only choose one of the active policies. In case you select both of them, the average improvements of the s_Q and s_Q_perishable over the baseline policy will be shown.



Filters

Visual level filters

Average of % NR Increase...

PolicyID

is s_Q

Filter Type

Basic filtering

■	Select All	
☑	s_Q	660
☐	s_Q_perishable	660
☐	Sim	1320

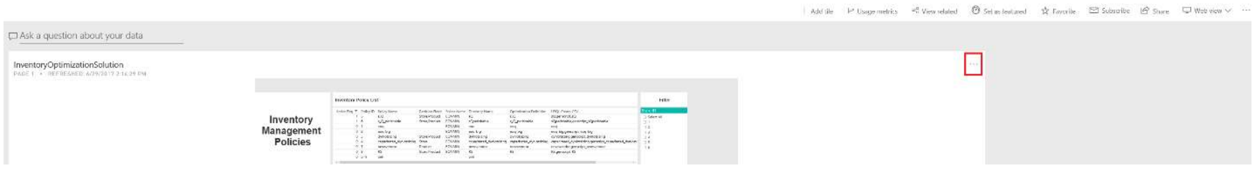
[Optional] Publish the dashboard to Power BI online

Note that this step requires a Power BI account (or Office 365 account).

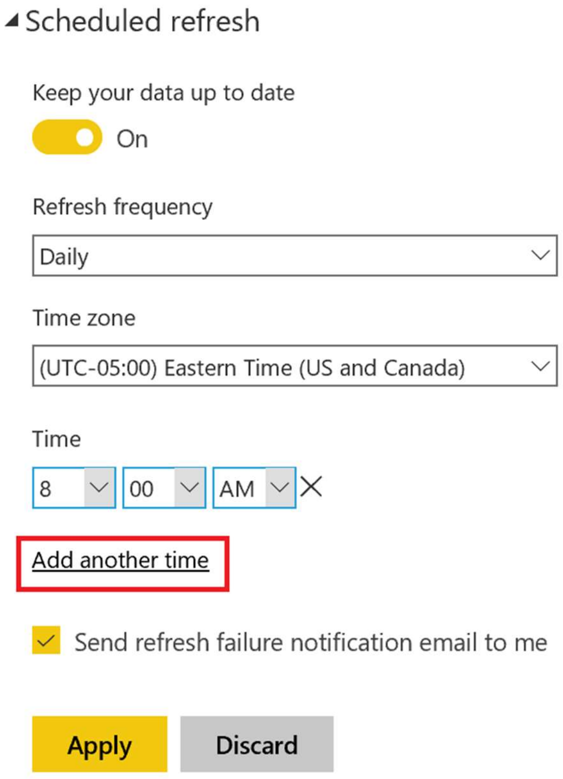
- Click **Publish** on the top panel. Choose **My workspace** (or any other workspace where you wish to publish the dashboard) and after a little while you will see a window showing that publishing to Power BI is succeeded.
- Click the link "Open 'InventoryOptimizationSolution.pbix' in Power BI" in the window to open the dashboard in a browser. You should see the dashboard in the browser. If not, click 'InventoryOptimizationSolution' under **REPORTS** section of **My Workspace** in the navigation

panel on the left-hand side. You may need to click the arrow sign besides **My Workspace** and scroll down to find **REPORTS** section.

- Click **Pin Live Page** on the top. On the pop-up window, choose **New Dashboard**, and put the name of the new dashboard, e.g. InventoryOptimizationSolution, and click **Pin Live**.
- Click **InventoryOptimizationSolution** in the **DASHBOARDS** section of **My Workspace**. Click the three dots on the top-right of the dashboard tile (see the red box in the screenshot below). You may need to scroll to the right to see these three dots. Click the middle pencil icon to edit the tile details. In **Functionality**, check **Display last refresh time**, and click **Apply**. You will see the last refresh time showing up on the top-left of the dashboard.


Add tile
Usage metrics
View related
Set as featured
Favorite
Subscribe
Share
Web view
Ask a question about your data
InventoryOptimizationSolution
Inventory Management Policies

- Scroll down the bar on the right-hand side of **My Workspace** to find **InventoryOptimizationSolution** in the **DATASETS** section. Right click the dots besides **InventoryOptimizationSolution** and select **SCHEDULE REFRESH**. You might see a message "Your data source can't be refreshed because the credentials are invalid. Please update your credentials and try again." Please click *each* **Edit credentials** below this message and sign in your Office 365 account. Once this is done, you can click **Scheduled refresh** and choose **On** under **Keep your data up to date**. Then, you can specify the refresh frequency and select times for automatic data refresh by clicking **Add another time**.



◢ Scheduled refresh

Keep your data up to date

On

Refresh frequency

Daily

Time zone

(UTC-05:00) Eastern Time (US and Canada)

Time

8 00 AM

Add another time

✓ Send refresh failure notification email to me

Apply Discard

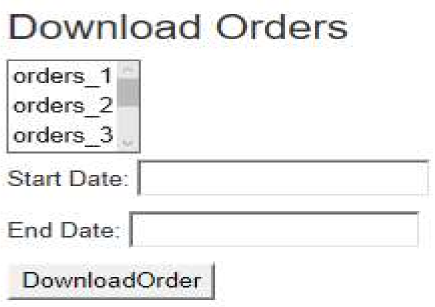
Website Guide

This is an optional part. If you are inventory manager and would like to review the suggested optimized price for your products, you can use this website to download orders. This section will explain how to use the website. If you want to deploy the website, click here. The website provides two functionalities to the user/Inventory Manager. This is a demo website basically to illustrate on how the data on Azure DataLake Store can be accessed through web application.

1. Download and Review Orders

User can get all the orders generated by the optimizer in csv format when searched between a date range. For this:

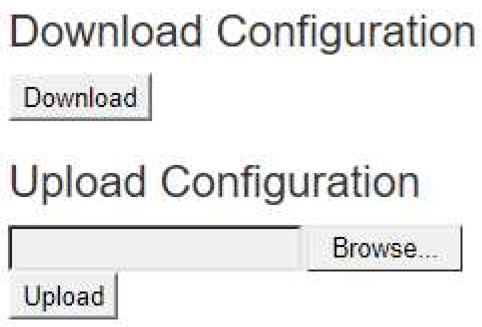
- Go to the website which is deployed after following the instructions here.
- Select the order numbers (order number is the store number) from the dropdown list. To select orders for multiple store, hold control and click on the all the store you want to select.
- Select Start and End date.
- Click DownloadOrder.
- This will download a csv file with the optimized order results for the selected store and the date range. The download may take time depending upon the number of stores and days.



2. Download and Upload Configuration file

User can download the configurations used by the optimization algorithm. Users can change the parameters in the configuration file as per their need. The instructions on understanding and changing configurations are provided under **Update Configuration File** in this technical guide.

- Click on Download Configuration.
- It will download a xls file.
- Once you have updated the configuration file, upload it back by browsing the file in upload section and clicking Upload.



Structure of Directories in Data Lake

In this section, we explain the content of each directory in Azure Data Lake Store.

configuration

- Configurations.xlsx: configurations of inventory policies. See the section Structure of configuration file for details.
- InventoryPolicy.txt: a subset of inventory policy configurations for Power BI visualization.

inventory_scripts

- download_scripts.py: This script is included in the Docker image provided. It downloads the following four scripts to the Docker container at the beginning of Azure Batch task execution
  - <inventory policy>.py: a Python script for the specified inventory policy located under the folder of the inventory policy. This script contains Pyomo code formulating optimization problems
  - inventory_optimization_task.py: A python script for formulating and solving optimization problems.
  - mipcl_wrapper.py: A Pyomo wrapper for MIPCL solver, as Pyomo doesn't fully support MIPCL.
  - upload_to_adls.py: This script is invoked after inventory_optimization_task.py and uploads task logs to ADLS.
- A folder for each inventory policy. Each folder contains the following three scripts specific for the inventory policy.
  - <inventory policy>.py: A Python script using Pyomo to formulate optimization problem for the specific inventory policy.
  - <inventory policy>.usql: A U-SQL script that extracts known values of optimization problems from the raw data.
  - order_<inventory policy>.usql: A U-SQL script that uses optimization problem solution, current time and inventory levels to place new orders.

rawdata: demand forecasts, inventory level, and sales data generated by data simulator. See the Data Schema section for details.

optimization

- input_csv folder: csv input to inventory optimization generated by USQL script <inventory policy>.usql. The folder contains a sub folder for each inventory policy. Each sub folder contains a sub folder for each data partition, e.g. <store, product>.
- output_csv folder: inventory optimization solution generated by script inventory_optimization_task.py. The folder contains a sub folder for each inventory policy. Each sub folder contains a sub folder for each data partition, e.g. <store, product>.
- log folder: log files of Azure Batch task (inventory_optimization_task.py) execution. The folder contains a sub folder for each inventory policy. Each sub folder contains a sub folder for each execution date time. For each data partition, there are two files: opt_<partition>_stderr.txt and opt_<partition>_stdout.txt. When the optimization tasks finish successfully, the stderr files should be empty.

orders: See the Data Schema section for more details.

- Sim folder: orders of baseline policy generated by data simulator.
- <inventory policy> folder: orders generated by the GenerateOrder web job by executing order_<inventory policy>.usql on Azure Data Lake Analytics.
- metric.csv: key performance metrics of inventory management policies.
- summary_metric.csv: summary of performance metrics used by Power BI visualization.
- inventory.csv: stores the inventory level of every product at the end of each day given the store ID and inventory management policy. It is generated based on the partial order files by the Evaluation web job. The inventory information is used for computing the turnover ratio and visualized in the inventory level section of the Power BI dashboard.

privateparameters: Internal data of simulator.

publicparameters: Static meta data of stores, departments, products, etc. See the Data Schema section for more details.

webjob_log

- webjob_generate_order: Output log of the GenerateOrder web job.
- webjob_optimization: Output log of the InventoryOptimization web job.
- ComputationTime.csv: Total running time of the Main web job.
- LastSimulationDateTime.txt: Last simulation date time in demo mode. See the Switching from Demo to Production Mode section for more details.

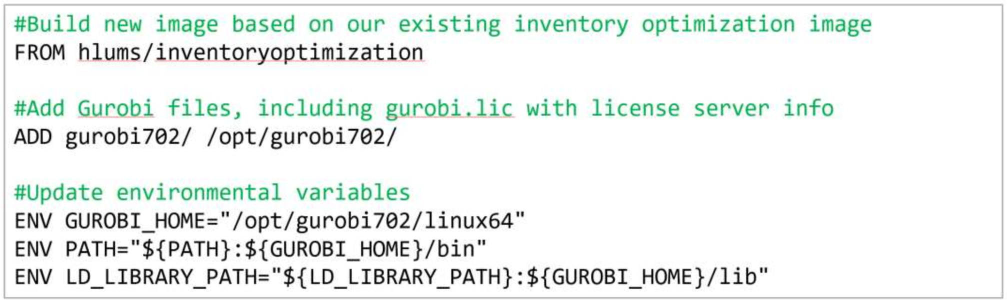
How to Integrate New Solvers

In this solution, we provide a Docker image with the Bonmin solver included. This Docker images is based on continuumio/anaconda3:4.2.0 with Python 3.5. You can add additional open-source solvers like MIPCL and commercial solvers like Gurobi.

To add new solvers to the Docker image, you first need to learn some basics of Docker. The official Docker website provide very comprehensive documentation. Going through the **Get started** section should be sufficient for adding new solvers to this solution. Below are the major steps to update the Docker image.

- Install Docker on your computer
- Pull the existing Docker hlums/inventoryoptimization from the Docker Hub
- Add the new solver to the Docker image. There are two ways to add a new solver to the Docker image.
  Option 1:
  1) Create a container from the Docker image provided.
  2) Copy the new solver files to the container. You may need to install some additional dependencies needed by the solver.
  3) Commit the container to a new Docker image.

  Option 2: Write a Docker file and build a new Docker image using the Docker file. Below is an example Docker file for adding Gurobi to the Docker image we provide.



```
#Build new image based on our existing inventory optimization image
FROM hlums/inventoryoptimization

#Add Gurobi files, including gurobi.lic with license server info
ADD gurobi702/ /opt/gurobi702/

#Update environmental variables
ENV GUROBI_HOME="/opt/gurobi702/linux64"
ENV PATH="${PATH}:${GUROBI_HOME}/bin"
ENV LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${GUROBI_HOME}/lib"
```

- Push the new Docker image to Docker Hub or your private container registry, e.g. Azure Container Registry.
- Update the value of **DOCKER_REGISTRY_IMAGE** in the Application settings of your web app. Replace the original Docker image name with your new Docker image name.
- If you are using a private container registry, you also need to add the following three additional Application settings
  DOCKER_REGISTRY_SERVER
  DOCKER_REGISTRY_USERNAME
  DOCKER_REGISTRY_PASSWORD
  You can find these values from your private container registry.
- Add the new solver to Configurations.xlsx
  1) Navigate to portal.azure.com
  2) On the left tab click **Resource Groups**
  3) Click on the resource group of the deployed solution
  4) Click the **Data Lake Store** in the list of resources.
  5) On the left side, click **Data Explorer**
  6) Navigate to the folder **configuration**. Click on the **. . .** on the right side of the file **Configurations.xlsx** and select **Download**.
  7) Open **Configurations.xlsx** and navigate to the **SolverConfig** tab. Add the SolverName, SolverPath, and DefinitionFileExtension of the new solver. The SolverPath is the absolute path of the solver in the Docker image. The DefinitionFileExtension is the extension of the input file to the solver. Then navigate to the I**nventoryPolicyConfig** tab. Update the SolverName column of the inventory policies you want to use the new solver. Save you changes and close the file.

Note: We use Pyomo as an interface with the solvers. Different solvers may work with Pyomo in slightly different ways. For Gurobi, Pyomo doesn't need the absolute path of the solver, so please enter *default* in the SolverPath column. For other solvers, please test with Pyomo and enter proper values in the SolverPath column. You can also modify the script **/inventory_scripts/inventory_optimization_task.py** in Azure Data Lake Store to meet the requirement of a particular solver.

8) Return to the **configuration** folder in Azure Data Lake Store.
    - Click **Upload** at the top.
    - In the **Upload files** blade, click the folder icon on the right to browse on your computer and select the updated **Configurations.xlsx** file.
    - Check **Allow overwrite existing files**.
    - Click **Add selected files**.

Additional steps for commercial solver

If you are adding a commercial solver, you will need to set up a license server and add some additional Application settings. Here we use Gurobi as an example.

First, you need to obtain a Gurobi floating license and set up a license server.

- Get a Gurobi floating license and download Gurobi from the official website.
- Create an Azure Virtual Machine (VM) to be used as the license server. You can find instructions on starting an Azure VM here. Note this VM should be created in the same region where you will create your Azure Batch pools, as the license requires the license server and client to be in the same virtual network.
- Copy the Gurobi files to the Azure VM.
- Retrieve Gruobi license by running ***grbgetkey <license key>*** in the bash shell of the license server.
- Start the license server by running ***grb_ts.***
- Create a gurobi.lic file with the license server info in the following format
  TOKENSERVER=<license server name>
  This file should be included in the Docker image as discussed above.
- For more details about setting up Gurobi license server, see instructions here and here.

Next, you need to create a User Subscription Batch Account. The batch account created in this solution is the traditional Batch Service type. A User Subscription Batch Account allows specifying a virtual network when creating the Batch pool. See instructions on creating the account here. Note that this account should be created in the same resource group with the other resources in the solution, so that the same Service Principle can be used to access the new batch account.

Last, add the following additional Application settings in your Web App.

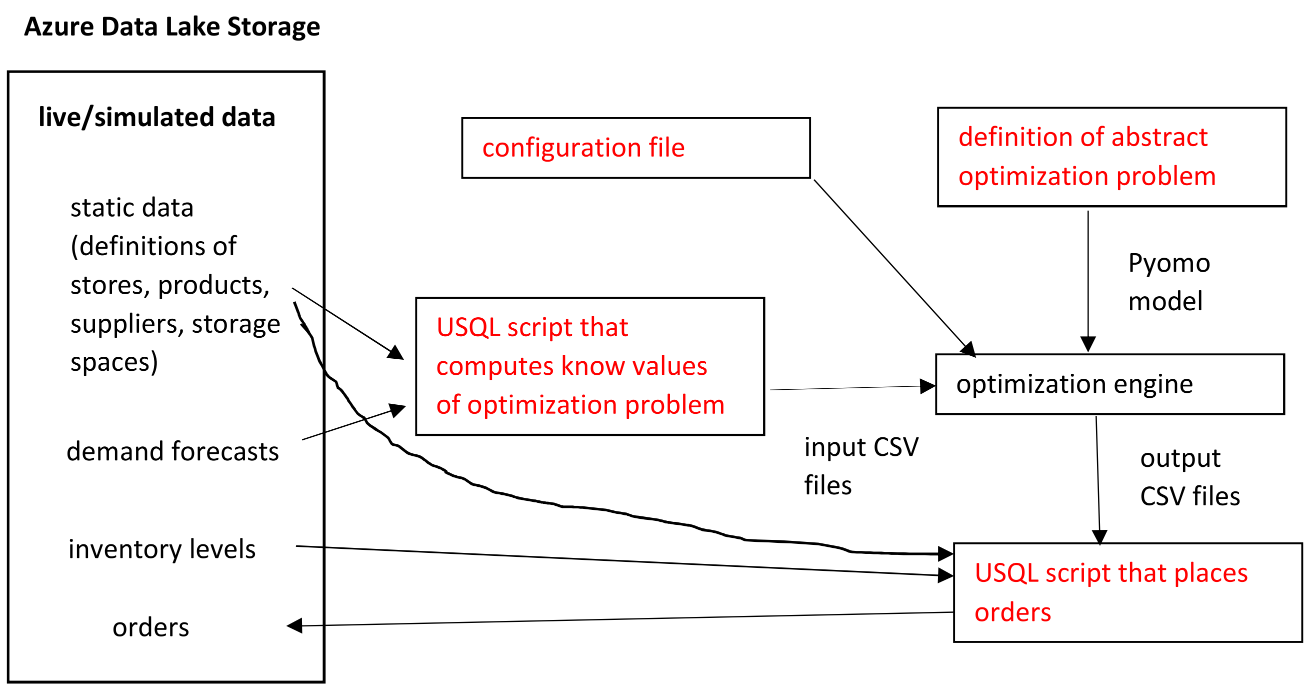
- VIRTUAL_NETWORK_NAME
- VIRTUAL_NETWORK_RESOURCE_GROUP
- VIRTUAL_NETWORK_SUBNET_NAME
- VIRTUAL_NETWORK_ADDRESS_PREFIX

To find the values of these properties

- Navigate to portal.azure.com
- On the left tab click **Resource Groups**
- Click on the resource group of the deployed solution
- Click the license server **Virtual machine** in the list of resources
- In the **Overview** page, click the link under **Virtual network/subnet**
- In the **Virtual network** page, click **Subnets**
- You can either use the **default** subnet or create new subnet as needed.

How to Define New Inventory Management Policies

In order to define a new inventory management policy a data scientist needs to define abstract optimization problem using Pyomo library and write two USQL scripts that compute known values of optimization problem and place orders. Finally, data scientist needs to update configuration file. We describe these steps below. The following diagram shows the connections between scripts and the data:


Azure Data Lake Storage
live/simulated data
static data (definitions of stores, products, suppliers, storage spaces)
demand forecasts
inventory levels
orders
configuration file
definition of abstract optimization problem
Pyomo model
USQL script that computes know values of optimization problem
optimization engine
input CSV files
output CSV files
USQL script that places orders

In this diagram the component that should be provided by data scientist are marked with red color. The optimization engine is part of our solution and should not be changed when defining new policies unless the data scientist needs to use a new solver to solve optimization problem of a newly defined policy. See section **How to Integrate New Solvers** for instructions how to plug in a new solver into optimization engine.

In the next sections we describe in more detail various components of this diagram.

Define Abstract Optimization Problem using Pyomo

Our solution uses Pyomo library to define abstract optimization problems. See [documentation page of Pyomo](#) for an introduction to Pyomo library and basic examples.

Section Sample Optimization Problems lists optimization problem that we provide with this solution. Pyomo implementation of these problems can be found in Python files under /inventory_scripts directory. Python file with Pyomo code has the following template:

```
from pyomo.environ import *
def defineOptimization():
    model = AbstractModel()

     Pyomo code that defines variables, known values, constraints,
    objective function

    return model
```

The name of the function in the above code should not be changed.

Write USQL Script that Computes Known Values of Optimization Problem

We use USQL scripts to compute known values of optimization problems. These scripts read raw data that is stored in Azure Data Lake Storage and create multiple CSV files with the known values of optimization problems. Section **Data Schema** describes the schema of the data that is an input to these USQL scripts. The data in CSV files is used to instantiate an abstract Pyomo model, which in turn is an input to optimizer.

USQL scripts are executed by Azure Data Lake Analytics service. Directory /inventory_scripts has examples of the USQL scripts for the sample optimization problem provided with our solution: sQ.usql, sQperishable.usql, eoq.usql, eoq_big.usql, dynlotsizing.usql, capacitated_dynlotsizing.usql, newsvendor.usql and RS.usql.

Directory /optimization/input_csv has examples of CSV files that are generated by USQL scripts. Each CSV file has one of the following three formats:

1) Definition of a single coefficient:

```
 Coefficient name
value
```

The name of the file should follow the template
<policy directory name>_<file id>_P_0<partition id>.csv

2) Definition of (k-1)-dimensional array of coefficients:

```
Value name 1,Value name 2,...,Value name k-1,Value name k
value1,1, value1,2, ..., value1,k-1, value1,k
value2,1, value2,2, ..., value2,k-1, value2,k
...
valuen,1, valuen,2, ..., valuen,k-1, valuen,k
```

The name of the file should follow the template
<policy directory name>_<file id>_P_<k-1><partition id>.csv

3) Definition of the set of values

```
Set name
value 1
value 2
 ...
value k
```

The name of the file should follow the template
<policy directory name>_<file id>_S_0<partition id>.csv

<policy directory name> is the directory name of the policy specified in configuration file (see section **Structure of Configuration File** for more details about the configuration file).

In all file names <file id> is an integer. The files are loaded in the increasing order of their <file id>. Hence if the values in file A depend on the values in file B then file B should have lower <file id> than file A. An example of such dependency is when B specifies number of elements in array (e.g. 5) and file A lists all indices and elements of the array.

See section **Structure of Configuration File** for the description of the concept of partitions. If inventory management policy does not use partitions, then <partition id> is an empty string. All files in the same partition with r levels have the following <partition id>:

<partition value 1><partition value 2>_ ... _<partition value r> ,

For example, if inventory policy is partitioned by store and product then r=2, <partition value 1> is store ID and <partition value 2> is product ID.

If inventory management policy uses partitions, then USQL script should generate multiple sets of inputs, one for each partition. This is achieved by adding to first USQL script a code that generates a second USQL script. The first USQL script (e.g. sQ.usql) creates a table in Azure Data Lake Analytics that has known values of optimization problems across all partitions. The second USQL script reads this table and distributes it across multiple sets of CSV files, one set per each partition. See an example of this technique in the implementation of sQ policy, files /inventory_scripts/sQ/sQ.usql and /inventory_scripts/sQ/genscript_sQ.usql. The latter file is generated by the former one.

If inventory management policy does not use partitions then generated CSV files should be placed in /optimization/input_csv/<policy_directory_name>. If inventory management policy uses partitions then the generated files for each partition should be placed in /optimization/input_csv/<policy_directory_name>/<partition directory> , where <partition directory> with r levels of partition has the structure

<partition value 1>/<partition value 2>/.../<partition value r> .

Write USQL Script that Places Orders

We use USQL scripts to generate orders. These scripts read solutions of optimization problems in Azure Data Lake Storage and updates multiple CSV files with the orders, one file per store. These files have names /orders/<policy directory name>/orders_<store ID>.csv . The schema of these files is described in section **Data Schema**.

Directory /orders has examples of order files that are generated by USQL scripts.

Directory /inventory_scripts has examples of the order generating USQL scripts for the sample optimization problem provided with our solution: order_sQ.usql, order_sQperishable.usql, order_eoq.usql, order_eoq_big.usql, order_dynlotsizing.usql, order_capacitated_dynlotsizing.usql, order_newsvendor.usql and order_RS.usql.

Since ordering USQL file should output data to multiple files, one file per store, we use the same technique of running two USQL scripts as in the previous section. The first USQL script (e.g. order_sQ.usql) creates a table in Azure Data Lake Analytics that has new orders across all stores. The second USQL script reads this table and distributes it across multiple CSV files, one file per each store. See an example of this technique in the implementation of sQ policy, files /inventory_scripts/sQ/order_sQ.usql and /inventory_scripts/sQ/gen_order_sQ.usql. The latter file is generated by the former one.

The newly generated orders are appended to the existing orders in the files /orders/<policy directory name>/orders_<store id>.csv and /orders/<policy directory name>/partial_orders_<store id>.csv

Update Configuration File

When defining a new inventory management policy, a data scientist needs to add a line in configuration file that defines various properties of the policies. See section Structure of Configuration File for the detailed description of the format of configuration file. Note that if the policy uses partitions and two USQL scripts to create CSV files then the names of the both USQL scripts should be specified in the configuration file. Similarly, if the policy generates orders for multiple stores and uses two USQL scripts for that purpose, then the names of the both USQL scripts should be specified in the configuration file.

Data Schema

In this section we describe the schema of the data generated and consumed by simulator. The real data coming from external sources should follow this schema. All datasets are stored in Azure Data Lake Storage. The data is divided into two categories, static and dynamic. Static data doesn’t change over time and include various properties of stores, products and suppliers. Dynamic data changes over time and includes sales, inventory levels, orders and demand forecasts.

Static Datasets

Static datasets are stored in /publicparameters folder.

Store

This dataset stores information about stores, one line per store. The dataset is located at /publicparameters/stores.csv .

Column	Type	Description	Sample Value
StoreID	string	Store ID	1
StoreName	string	Store name	Store 1
AvgHouseHoldIncome	float	Average annual household income of store customers (this column is not used by inventory optimization)	100000
AvgTraffic	float	Average number of visitors to the store in one day (this column is not used by inventory optimization)	60000

Store Storage

This dataset stores information about storage spaces in stores, one line per storage space in each store. The dataset is located at /publicparameters/store_storage.csv .

Column	Type	Description	Sample Value
StoreID	string	Store ID	1
StorageID	string	Storage space ID	3
StorageName	string	Name of the storage space	Refrigerators
StorageVolume	float	Volume of the storage space	100
StorageCostBudget	float	Maximal storage cost of all items stored in the storage at the same time	300

Store Departments

This dataset stores information about departments in stores, one line per department in each store. The dataset is located at /publicparameters/store_departments.csv .

Column	Type	Description	Sample Value
StoreID	string	Store ID	1
DepartmentID	string	Department ID	3
DepartmentName	string	Department Name	Bakery

Brands

This dataset stores information about brands sold in all stores, one line per brand. The dataset is located at /publicparameters/brands.csv .

Column	Type	Description	Sample Value
BrandID	string	Brand ID	1
BrandName	string	Brand name	Best Bread

Brands Products

This dataset stores information about products in each brand, one line per product in each store. The dataset is located at /publicparameters/brands_products.csv .

Column	Type	Description	Sample Value
BrandID	string	Brand ID	1
ProductID	string	Product ID	5
ProductName	string	Product name	White Bread
MSRP	float	Minimal suggested retail price	40
ProductVolume	float	Volume of the single unit of product	3
ShelfLife	string	Number of days/weeks/months that the product can be in the store until it expires. Empty string if the product does not expire.	“3 days”, “1 week”

Store Department Brand Products

This dataset stores information about products sold in the stores, one line per product in each store. The dataset is located at /publicparameters/store_department_brand_products.csv .

Column	Type	Description	Sample Value
StoreID	string	Store ID	1
DepartmentID	string	Department ID	2
BrandID	string	Brand ID	3
ProductID	string	ProductID	4
MSRP	float	Minimal suggested retail price	25
DisposalCost	float	Cost of disposal of a single item of the product	2

Store Product Storage

This dataset stores information about storage of products in each store, one line per product in each store and storage space. The dataset is located at /publicparameters/store_product_storage.csv .

Column	Type	Description	Sample Value
StoreID	string	Store ID	1
StorageID	string	Storage ID	2
ProductID	string	Product ID	3
StorageCost	float	Cost of the storing a single unit of the product	2
MissedSaleCost	float	Cost of missed sale opportunity (when there is a demand but no items in the storage)	4
MinInventorySize	int	Minimal number of items of the product that should be in the storage space	10
MaxInventorySize	int	Maximal number of items of the product that can be stored in the storage space	1000

Suppliers

This dataset has supplier data, one line per supplier. The dataset is at /publicparameters/suppliers.csv .

Column	Type	Description	Sample Value
SupplierID	string	Supplier ID	1
SupplierName	string	Supplier name	Jeans Supplier
ShippingCost	float	Cost of as single shipping (this is an additional cost to the shipping costs of individual products)	100
MinShippingVolume	float	Minimal volume of the order. -1 if there is no lower bound on the volume of the order.	20
MaxShippingVolume	float	Maximal volume of the order. -1 if there is no lower bound on the volume of the order.	50
FixedOrderSize	int	Exact number of items that should be in the order. -1 if there could be any number of items in the order.	500
PurchaseCostBudget	float	Maximal cost of the order. -1 if there is no upper bound on the cost of the order.	2000

Store Product Supplier

This dataset stores information about suppliers of products in each store, one line per combination of product and supplier in each store. The dataset is at /publicparameters/store_product_supplier.csv .

Column	Type	Description	Sample Value
StoreID	string	Store ID	3
SupplierID	string	Supplier ID	1
ProductID	string	Poduct Id	5
LeadTime	int	Lead time, in days	4
LeadTimeConfidenceInterval	int	Lead time confidence interval, in days (e.g. +/- 1 days)	1
MinOrderQuantity	int	Minimal number of items in a single order	100
MaxOrderQuantity	int	Maximal number of items in a single order	1000
QuantityMultiplier	int	Quantity multiplier. Order size = x * quantity multiplier, where x is a positive integer	10
Cost	float	Purchase cost of a single item	20
BackorderCost	float	Backorder cost of a single item	25
ShippingCost	float	Shipping cost of a single item	4
ShiupmentFreq	string	Maximal shipping frequency	“3 days”
ServiceLevel	float	Service level (fraction of demand that should be fulfilled)	0.99

Dynamic Datasets

Dynamic datasets are stored in /rawdata folder.

Demand Forecast

This dataset stores forecasted demand. The dataset is located at /rawdata/demand_forecasts . Forecasts for product with ProductID = Y that is sold in store with StoreID = X are stored in the directory /rawdata/demand_forecasts/X/Y/ . This directory has multiple files with the names <date>_00_00_00.csv , where <date> is the date when the forecast was made. The dataset has the following schema:

Column	Type	Description	Sample Value
StoreID	string	Store ID	3
ProductID	string	Product ID	1
DateTime	DateTime	Future date	2017-07-15
Demand	int	Forecasted mean value of demand at the date in DateTime column	24
PredictedDemandDistribution	string	Predicted distribution of demand. Empty string if the predicted demand is deterministic or doesn't follow Poisson or Normal distribution	"Poisson", "Normal"
PredictedDemandVariance	float	Variance of the distribution of demand. -1 if the variance is not known.	1.5
PredictedDemandProbability	float	Predicted probability of demand. 1 if predicted demand is deterministic.	0.3

Demand Forecast dataset can store deterministic and probabilistic forecasts. Deterministic forecasts will have one row per DateTime, PredictedDemandVariance=-1 and PredictedDemandProbability=1. Probabilistic forecasts can be specified in two ways:

- Implicit - specifying type distribution, "Poisson" or "Normal" in PredictedDemandDistribution, specifying mean and variance in Demand and PredictedDemandVariance.
- Explicit – specifying all probability mass points Pr(forecast=Demand) = PredictedDemandProbability.

In deterministic and implicit probabilistic forecasts the dataset will have one line per StoreID, ProductID and DateTime. In explicit probabilistic forecast the dataset will have multiple lines per StoreID, ProductID and DateTime, but one line per StoreID, ProductID, DateTime and PredictedDemandPRobability.

Sales

This dataset stores sales data. The dataset is located in /rawdata and consists of multiple files with names sales_store<ID>_<date>_00_00_00.csv , ID is the ID of the store and <date> is a sales date. Each such file has all sales of a store in a single day. The dataset has one line per sales transaction and has the following schema:

Column	Type	Description	Sample Value
StoreID	string	Store ID	4
ProductID	string	Product ID	2
TransactionDateTime	DateTime	Transaction timestamp	2017-04-23 15:06:25
Units	int	Number of units sold	1
Price	float	Sale price of a single unit	2.5

Inventory Levels

This dataset stores inventory levels at the end of each day. The dataset is located in /rawdata and consists of multiple files with names inv_store<ID>_<date>_00_00_00.csv , ID is the ID of the store and <date> is the date of counting inventory. The dataset has the following schema:

Column	Type	Description	Sample Value
StoreID	string	Store ID	1
ProductID	string	Product ID	5
InventoryDateTime	DateTime	Date of counting inventory	2017-06-02
Units	int	Number of units in inventory	54
ExpiryDatetime	DateTime	Expiration date	2017-09-04 00:00:00

Orders

This dataset stores orders that are generated by inventory management policies. Each policy has its own directory of orders, located at /orders/<policy directory name>, where <policy directory name> is specified in configuration file. Order files have names orders_<StoreID>.csv , where StoreID is ID of the sore where inventory policy manages products. The dataset has the following schema:

Column	Type	Description	Sample Value
InventoryPolicyName	string	Policy name (as defined in configuration file)	sQ
StoreID	string	Store ID	1
ProductID	string	Product ID	2
SupplierID	string	Supplier ID	3
Quantity	int	Order quantity	20
OrderTimestamp	DateTime	Order date and time	01/05/2017 00:00:00
ETA	DateTime	Estimated time of arrival	01/08/2017 00:00:00
ConfidenceInterval	int	Estimated time of arrival confidence interval, in days (e.g. +/- 1 days)	1
Fulfilled	boolean	Indicator if the order is delivered to the store	True, False

Integration with External Sources

In this solution, we use Azure Data Lake Store (ADLS) for storing input raw data, intermediate data, and final results. This is because Azure Data Lake Store is very flexible in terms of data type, size, and ingestion speed. In addition, ADLS includes enterprise level security and management capabilities.

If your data reside in a different storage system or you want to move the final results to the storage of your ordering system, you can use Azure Data Factory to move data to and from Azure Data Lake Store. For details of supported storage systems and data movement with Azure Data Factory, please see the documentation here.

Azure Data Lake Store is also compatible with many open-source big data applications like Spark and Storm. You can find a full list of supported open-source applications here.

To work with Azure Data Lake using programming languages, e.g. Java and Python, please see the **Get started** section on this page.

Structure of Simulator

In this solution we provide a web job that simulates operations of chain of stores, including sales, demand forecasting, placing orders and receiving deliveries from the past orders. The simulator has many parameters that are defined in the beginning of the code (Simulator.py file in Simulator web job). The table below summarizes the most important ones:

Parameter	Description	Default value
n_stores	number of stores	6
n_brands	number of brands	20
n_departments	number of departments in each store	4
n_suppliers	number of suppliers	5
products_per_brand	number of products in each brand	1
n_storage_spaces	number of storage spaces in each store	4

Simulator has the following assumptions:

- all stores have the same departments
- all stores sell the same brands and products
- all stores have the same number of storage spaces
- all suppliers deliver products to all stores
- all brands have the same number of products
- all stores are open every day, from 7am till 9pm
- each sale transaction has only one item
- there are no special days (weekends, holidays) with predicted significant spike or drop of demand.

Although all stores have a similar configuration, predicted and actual demand, as well as the sizes of storage spaces, vary across the stores. At each call the simulator simulates operations of all stores in one day:

1. Beginning of the day, before opening the store:
   - predict demand for the next 6 weeks
   - receive deliveries from previous orders of active policies and update the current inventory.
2. Opening hours: simulate actual demand and sales. The sale transaction is generated if the requested product is available in inventory. After the sale happens, the inventory for the sold product is updated.
3. End of the day, after closing the store:
   - remove expired products from inventory
   - place orders of the Sim policy (see description of this policy below).

In addition to simulating store operations, the simulator also implements a special "Sim" policy, that is used as a baseline when evaluated performance of optimized policies. Sim policy places an order when there is a request for sale of the product. The order size is one item and it is placed regardless of the current inventory level. Sim policy is never active and its orders are never delivered.

Simulation of predicted and actual demand

In this section we describe the technical details of simulating predicted and actual demand.

Predicted and actual demand depend on the price of the product. The simulated prices of products vary across the stores and depend on MSRP and the cost of the product. The cost of the same product varies across stores and MSRP of product in store is defined as MSRP_multipler * cost, where MSRP_multiplier is a random number between 1.1 and 1.5. The price of the product in each store is drawn from normal distribution with mean and standard deviation that depend on MSRP and cost:

$$\text{Price(Product,Store)} \sim \text{Normal}\left(\text{mean=cost} + 0.8 \cdot (\text{MSRP-cost}),\ \text{std=}\left(0.5{\cdot}(\text{MSRP-cost})\right)^2\right).$$

Let Department(Product,Store) be a set of products sold in the department where the given product is sold. The relative price of the product is

$$\text{RelPrice(Product,Store)} = \frac{\text{Price(Product,Store)}}{\sum_{\text{Product}'\in\text{Department(Product,Store)}} \text{Price(Product}',\text{Store)}}$$

and the relative discount over MSRP is

$$\text{RelDiscountOverMSRP(Product,Store)=}\frac{\text{MSRP-Price}}{\text{Price}}.$$

Predicted daily demand of the product has Poisson distribution with the mean value depending on the relative price and the relative discount over MSRP:

$$\text{PredictedDailyDemand(Product,Store)} \sim$$
$$\text{Poisson}\left(\frac{(\text{RelPrice - 1}) \cdot \text{Price} \cdot \text{PriceElasticity(Product)} + \dfrac{\text{AvgTraffic(Store)} \cdot \text{Desirability(Brand)}}{\text{1 - RelDiscountOverMSRP}}}{\text{RelPrice}^2}\right)$$

where

PriceElasticity(Product) - the price elasticity of the product, a random number between -1.3 and – 0.7. Each product has the same price elasticity in all stores.
AvgTraffic(Store) – average number of visitors in the store, drawn from a normal distribution with mean 100 and standard deviation 100.
Desirability(Brand) – a random number between 0.7 and 1.3 that expresses desirability of product's brand.

We assume that there are no dependencies between demands of different products. We simulate the actual demand by sampling demand events from Poisson distribution of predicted demand. Poisson distribution of daily demand of all products implies that for each hour when the store is open, the time between the actual demand events has exponential distribution with the mean inversely proportional to predicted demand:

$$\Delta t(\text{Store}) \sim \exp\left(\frac{\text{number of hours the store is open}}{\sum_{\text{product' in store(product)}} \text{PredictedDailyDemand(Product',Store)}}\right)$$

where store(product) is a set of products in the store where a given product is located.

We draw the time to the next demand event from the above exponential distribution. When the demand event occurs, the probability that the demand event is for a given product is

$$\frac{PredictedDailyDemand(Product, Store)}{\sum_{product' \ in \ store(product)} PredictedDailyDemand(Product', Store)}$$

Sample Inventory Management Policies

In this section we describe 7 sample inventory management policies that are provided with our solution. The first two policies, (s,Q) policy and (s,Q) policy for perishable items are active after initial deployment. The rest of the policies are not active after the initial deployment, but can be activated by following instructions in section **How To Define New Inventory Management Policies**.

(*s*, *Q*) policy

Definition

(*s,Q*) policy manages individual products. In this policy the inventory level of a product is monitored continuously. When the inventory level decreases to *s* items, an order for *Q* items is placed. The optimal values of *s* and *Q* vary between products and are found by solving optimization problem.

(*s,Q*) policy has multiple assumptions:

- product does not expire

- the lead time is much smaller than the time required to sell *Q* items
- the lead time is known and is deterministic
- there is a probabilistic forecast of demand for the product
- the demand for product is close to constant and does not change much over time
- shipping, holding and backorder costs of a product are known.

Examples of products that can be managed by (*s,Q*) policy are office supplies, auto parts and furniture.

Optimization problem for generating the policy

We use the following notation:

L – lead time (in days)
a – average demand in one day
μ=aL – average demand in the lead time
h – holding cost per day
K – cost of a single shipping
π – cost of a single backorder
X – demand during the lead time. This demand is a discrete random valuable with probability mass function Pr(*X*). Pr(*X*=*x*) is a probability of demand for exactly *x* items during the lead time.

The values of *s* and *Q*, found by optimization problem, minimize holding, shipping and backordering costs. The expected holding cost per day is

$$h\left(\frac{Q}{2}+s-\mu\right) \qquad (1)$$

the expected shipping cost per day is $\frac{Ka}{Q}$ and the expected backordering cost is

$$\frac{\pi a}{Q}\sum_{x=s}^{\infty}(x-s)\Pr(X=x) \qquad (2)$$

See [1] for the derivation of these quantities. The objective function of optimization problem is a sum of these three costs. The final optimization problem of (*s*,*Q*) policy is

$$\min_{s,Q}\ h\left(\frac{Q}{2}+s-\mu\right)+\frac{Ka}{Q}+\frac{\pi a}{Q}\sum_{x=s}^{\infty}(x-s)\Pr(X=x)$$
$$\text{such that } s \text{ and } Q \text{ are non-negative integers.}$$

[1] P. Jensen and J. Bard. Operations Research Models and Methods. Wiley, 2002. Section 25.5. Available at www.me.utexas.edu/~jensen/ORMM/supplements/models/inventory/sq_policy.pdf

Implementation of optimization problem in Pyomo

In the backordering cost (1) the sum goes from s to infinity, where s is unknown variable. Since Pyomo does not support infinite sums, we replaced infinity with the upper bound *max_demand* on the demand during the lead time. This upper bound is a maximal value of X that has non-zero probability. Also, Pyomo does not support sums where the boundary is an optimized variable. We can rewrite (2) as

$$\frac{\pi a}{Q}\sum_{x=\mu+1}^{max\_demand} I(x \geq s)\cdot(x-s)\Pr(X=x)$$

where *I(x≥s)* is an indicator function, *I(x≥s) = 1* if *x ≥ s* and *0* otherwise. Furthermore, Pyomo does not support indicator functions. We approximate indicator function *I(x≥s)* by sigmoid function:

$$I(x \geq s) \approx \frac{1}{1+\exp(-d\cdot(x-s))} \tag{3}$$

where d is a positive constant. We use d=1000. The final optimization problem that is implemented using Pyomo library is:

$$\min_{s,Q}\ h\left(\frac{Q}{2}+s-\mu\right)+\frac{Ka}{Q}+\frac{\pi a}{Q}\sum_{x=\mu+1}^{max\_demand}(x-s)\Pr(X=x)\cdot\frac{1}{1+\exp(-1000\cdot(x-s))}$$

such that s and Q are non-negative integers.

This is mixed integer non-linear optimization problem. We use Bonmin solver to solve it.

(s, Q) policy for perishable items

Definition

(*s,Q*) policy described in the previous section assumes that the products have no expiration date. This assumption does not hold for many products, for example food and seasonal products. (*s,Q*) policy for perishable items manages such products by taking into account their expiration date. This policy places the orders in exactly the same way at the original (*s,Q*) policy.

We define the life time of the product as the time from getting the product into store till expiration. The assumptions of (*s,Q*) policy for perishable items are:

- product has expiration date
- life time of the product is fixed and does not change over different delivery days
- the cost of disposal of expired unit of product is known
- all assumptions of the original (*s,Q*) policy except the first one.

Optimization problem for generating the policy

We use the same notation as in the original (*s,Q*) policy. We also use the following terms:

W – cost of disposal of one expired unit
Y – demand during the lead time + life time. This demand is a discrete random valuable with probability mass function Pr(*Y*). Pr(*Y=y*) is a probability of demand for exactly *x* items during the lead time + life time.

The values of *s* and *Q*, found by optimization problem, minimize holding, shipping, backordering and disposal costs. The expected holding cost per day is the same as in the original (*s,Q*) policy (see (1)). The rest of the costs depend on the expected number of expired units:

$$ER(s,Q)=\sum_{y=-\infty}^{s+Q-1}(s+Q-y)\Pr(Y=y)-\sum_{y=-\infty}^{s-1}(s-y)\Pr(Y=y)\,. \tag{4}$$

The expected shipping cost per day is

$$\frac{Ka}{Q-ER(s,Q)},$$

the expected backordering cost per day is

$$\frac{\pi a}{Q-ER(s,Q)}\sum_{x=s}^{\infty}(x-s)\Pr(X=x) \tag{5}$$

and the expected disposal cost per day is

$$\frac{W\cdot a\cdot ER(s,Q)}{Q-ER(s,Q)}$$

See [2] for the derivation of the last four expressions. The objective function of optimization problem is a sum of the four costs. The final optimization problem of (*s,Q*) policy for perishable items is

$$\min_{s,Q}\left(K+\pi\sum_{x=s}^{\infty}(x-s)\Pr(X=x)+W\cdot ER(s,Q)\right)\cdot\frac{a}{Q-ER(s,Q)}+\mathrm{h}\cdot\left(\frac{Q}{2}+s-\mu\right)$$

such that s and Q are non-negative integers.

[2] H.N. Chiu. A good approximation of the inventory level in (*Q,r*) perishable inventory system. RAIRO – Operations Research, 33(1):29-45, 2010. Available at http://www.rairo-ro.org/articles/ro/pdf/1999/01/ro1999331p29.pdf .

Implementation of optimization problem in Pyomo

We use the same technique as in the original (*s,Q*) policy to replace infinite sums in (4) and (5) with finite ones and to eliminate the optimized variables from the boundaries of summation. Let d_{min} and d_{max} be minimal and maximal demand during lead time + life time respectively. We approximate the expected number of expired units by

$$ER'(s,Q)=\sum_{y=d_{min}}^{d_{max}}(s+Q-y)\Pr(Y=y)\cdot\frac{1}{1+\exp(-(s+Q-1-y))}-\sum_{y=d_{min}}^{d_{max}}(s-y)\Pr(Y=y)\cdot\frac{1}{1+\exp(-(s-1-y))}$$

The final optimization problem that is implemented using Pyomo library is

$$\min_{s,Q}\left(K+\pi\sum_{x=\mu+1}^{max\_demand}(x-s)\Pr(X=x)\cdot\frac{1}{1+\exp(-500\cdot(x-s))}+W\cdot ER'(s,Q)\right)\cdot\frac{a}{Q-ER'(s,Q)}+\mathrm{h}\cdot\left(\frac{Q}{2}+s-\mu\right)$$

such that s and Q are non-negative integers.

This is mixed integer non-linear optimization problem. We use Bonmin solver to solve it.

(R, S) policy

Definition

(*R,S*) policy manages individual products. In this policy the inventory level of a product is checked every *R* days. Let s be an inventory level at the time of inspection. After checking the inventory level, (R,S) policy places an order for *S* - *s* items. The optimal values of *R* and *S* vary between products and are found by solving optimization problem.

(*R,S*) policy has the same assumptions as (*s,Q*) policy.

(*R,S*) policy establishes a fixed shipping schedule for each product. Hence this policy can be used when a supplier cannot ship the product on-demand.

Optimization problem for generating the policy

We use the following notation:

L, *a*, *h*, *K*, *π* – the same definition as in (*s,Q*) policy.

X_R – demand during the lead time + review time R. This demand is a discrete random valuable with probability mass function Pr(X_R). Pr($X_R = x$) is a probability of demand for exactly *x* items during the lead time + review time *R*.

$\mu_{LR} = a(L+R)$ - average demand in the lead time + review time *R*.

The values of *R* and *S*, found by optimization problem, minimize holding, shipping and backordering costs. The expected holding cost per day is

$$h\left(\frac{aR}{2} + S - \mu_R\right)$$

the expected shipping cost per day is $\frac{K}{R}$ and the expected backordering cost is

$$\frac{\pi}{R}\sum_{x=S}^{\infty}(x-S)\,\mathrm{Pr}(X_R = x) \qquad (6)$$

See [3] for the derivation of these quantities. The objective function of optimization problem is a sum of these three costs. The final optimization problem of (*R,S*) policy is

$$\min_{R,S}\ h\left(\frac{aR}{2} + S - \mu_{LR}\right) + \frac{K}{R} + \frac{\pi}{R}\sum_{x=S}^{\infty}(x-S)\mathrm{Pr}(X_R = x)$$

such that R and S are non-negative integers.

[3] P. Jensen and J. Bard. Operations Research Models and Methods. Wiley, 2002. Section 25.7. Available at www.me.utexas.edu/~jensen/ORMM/supplements/models/inventory/rs_policy.pdf

Implementation of optimization problem in Pyomo

We use the same technique as in (*s,Q*) policy to replace the infinite sum in (6) with finite ones and to eliminate the optimized variables from the boundaries of summation. Let d_{min} and d_{max} be minimal and maximal demand during lead time + any possible review time respectively. The resulting approximate backordering cost is

$$\frac{\pi}{R}\sum_{x=d_{min}}^{d_{max}}(x-S)\Pr(X_R=x)\cdot\frac{1}{1+\exp(-10\cdot(x-S))}\ . \tag{7}$$

Notice that the probability mass function Pr($X_R = x$) is a function of R. For different values of R there will be different probability mass function. Internally, our Python code that defines input to optimization problem using Pyomo library, represents Pr($X_R = x$) as a 2-dimensional array p[R,x], where p[R,x] = Pr($X_R = x$). Pyomo does not support optimized variables as indices of arrays. To overcome this issue, we rewrite (7) as

$$\frac{\pi}{R}\sum_{r=1}^{R_{max}}\sum_{x=d_{min}}^{d_{max}}(x-S)\Pr(X_R=x)\cdot\frac{1}{1+\exp(-10\cdot(x-S))}\cdot I(r=R)\ ,$$

where R_{max} is the maximal possible value of R and I(r=R) is an indicator function, *I(r=R) = 1* if *r=R* and *0* otherwise. Since Pyomo does not support indicator functions, we approximate indicator function I(r=R) by a product of two sigmoid functions:

$$I(r=R)\approx I'(r=R)=\frac{1}{1+\exp(-d\cdot(r-R))}\cdot\frac{1}{1+\exp(-d\cdot(R-r))}\ ,$$

where d is a positive constant. We use d=10.

We also add a constraint $S - aL - aR \geq 0$ to kepp the holding cost $h\left(\frac{aR}{2}+S-\mu_{LR}\right)=h(S-aL-aR)$ nonnegative.

The final optimization problem that is implemented using Pyomo library is:

$$\min_{R,S}\ h\left(\frac{aR}{2}+S-\mu_R\right)+\frac{K}{R}+\frac{\pi}{R}\sum_{r=1}^{R_{max}}\sum_{x=d_{min}}^{d_{max}}(x-S)\Pr(X_r=x)\cdot\frac{1}{1+\exp(-10\cdot(x-S))}\cdot I'(r=R)$$

s.t. $S-aL-aR\geq 0$

R and S are non-negative integers.

This is mixed integer non-linear optimization problem. We use Bonmin solver to solve it.

EOQ policy

Definition

EOQ policy is one of the most popular inventory management policies and has many versions (see [4]). In this policy an order for Q items of the same product is placed every R days. The optimal values of *Q* and *R* vary between products and are found by solving optimization problem.

We implemented constrained lot sizing version of EOQ, described in Section 2.4 of [4]. This version of EOQ policy manages a set of products that are stored in the same storage space and generates orders such that the average storage costs of products in the storage space is less than a fixed threshold.

Constrained lot sizing EOQ policy has multiple assumptions:

- products do not expire
- the demand for each product is constant and does not change over time

- the lead time is very short and is negligible
- shipping cost of each product is known
- there is a limit on the storage expenses

Constrained lot sizing EOQ policy can be effective for products with flat demand and high storage costs. Examples of such products are grocery products that must be stored in refrigerators.

[4] J. Muckstadt and A. Sapra. Principles of Inventory Management. Springer, 2010. Section 2. Available at http://www.springer.com/us/book/9780387244921

Optimization problem for generating the policy

We use the following notation:

n – number of products managed by the policy
K_i – shipping cost of the i-th product
λ_i – predicted demand for the i-th product in the planning time period
C_i – storage cost of the i-th product in a single day
b – upper bound on the average storage cost in a single day
Q_i – order size of the i-th product.

The values of Q_i, found by optimization problem, minimize shipping costs subject to limited storage costs. The average daily shipping costs of all products are

$$\sum_{i=1}^{n} \frac{\lambda_i K_i}{Q_i}$$

and the average daily storage costs of all products are

$$\sum_{i=1}^{n} C_i \frac{Q_i}{2} \ .$$

See Section 2.4 of [4] for the derivation of these formulas. The final optimization problem of constrained lot sizing EOQ policy is

$$\min_{Q_1,\cdots,Q_n} \sum_{i=1}^{n} \frac{\lambda_i K_i}{Q_i} \tag{8}$$
$$s.t. \ \sum_{i=1}^{n} C_i \frac{Q_i}{2} \le b \tag{9}$$
$$Q_i \text{ is a nonnegative integer}, \ \ \forall\, i = 1,\ 2, \cdots,\ n$$

Implementation of optimization problem in Pyomo

Implementation of optimization problem (8) is very straightforward and does not require any additional transformations. This optimization problem is mixed integer nonlinear problem and is solved using Bonmin solver.

EOQ_big: Large-scale implementation of EOQ policy

When the number of items managed by constrained lot sizing EOQ policy becomes large (e.g. thousands), solving optimization problem (8) might require significant time (e.g. hours). To speed up the computation, we develop an equivalent linear programming problem.

It follows from the constraint (9) that Q_i is upper bounded by $m_i = \left\lceil \frac{2b}{C_i} \right\rceil$. Let

$$I_{ij} = \begin{cases} 1 \text{ if } Q_i = j \\ 0 \text{ otherwise} \end{cases}, \quad i=1,..,n, \quad j=1,...,m_i$$

be an indicator variable that tells if $Q_i = j$. The following optimization problem

$$\min_{I_{ij}} \sum_{i=1}^{n} \sum_{i=1}^{n} \frac{\lambda_i K_i}{j} I_{ij} \tag{10}$$

$$s.t. \sum_{i=1}^{n} \sum_{j=1}^{m_j} C_i \frac{j}{2} I_{ij} \leq b$$

$$\sum_{j=1}^{m_i} I_{ij} = 1, \quad i=1,...,n$$

$$I_{ij} \in \{0,1\}, \quad i=1,..,n, \quad j=1,...,m_i$$

is equivalent (8). The optimization problem (10) is linear and hence can be solved much faster than (8) by using fast mixed integer programming solvers. Because of the legal constraints, our solution does not contain any such solvers preinstalled. As a temporary workaround, optimization problem (11) can be solved using pre-installed Bonmin solver. In section **How To Integrate New Solvers** we describe how to plug into our solution two fast mixed integer programming solvers, a free solver MIPCL and a commercial solver Gurobi.

Pyomo implementation of optimization problem (10) is very straightforward and does not require any additional transformations.

Dynamic Lot Sizing policy

Definition

Dynamic lot sizing policy manages individual products. Dynamic lot sizing policy uses a planning time that is partitioned into the number of time periods of the same length (e.g. one week). Dynamic lot sizing policy can place and order for a product in the beginning of time period (e.g. on Monday). The size of the planned orders depends on the predicted future inventory and demand. Unlike (R,S) and EOQ policies, dynamic lot sizing policy does have any assumptions about the future demand and can take into account any patterns of future demand (e.g. increase of demand due to holidays and decrease of demand due to weather conditions). The optimal sizes of future orders are found by solving optimization problem.

Dynamic lot sizing policy has the following assumptions:

- product does not expire
- the lead time is very short and is negligible
- there is a deterministic forecast of demand for the product
- shipping and holding costs of a product are known
- suppliers can deliver unlimited number of items in a single shipment.

Examples of products that can be managed by dynamic lot sizing policy are office supplies. These products have changing demand patterns (e.g. low demand in June/July and high demand in August because of the new school year) and do not expire.

Optimization problem for generating the policy

We use the following notation:

h – storage cost during a time period

K – cost of a single shipping
T – number of time periods
d_i – predicted deterministic demand during the i-th time period
$\delta_i \in \{0,1\}$ – indicator if there will be an order at the beginning of the i-th time period
z_i – order size at the i-th time period. $z_i > 0$ if δ_i=1, 0 otherwise.

The values of δ_i and z_i minimize shipping and storage costs and are found by solving optimization problem. The total shipping cost over the planning time period is

$$K \sum_{t=1}^{T} \delta_t$$

and the total storage cost over the same time period is

$$h \sum_{t=1}^{T} \sum_{i=1}^{t} (z_i - d_i) \ .$$

The final optimization problem of dynamic lot sizing policy is

$$\min_{\delta_t, z_i} \quad \sum_{t=1}^{T} \left\{ K\delta_t + h \sum_{i=1}^{t} (z_i - d_i) \right\} \qquad (11)$$

$$\text{s.t.} \quad \sum_{i=1}^{t} z_i \geq \sum_{i=1}^{t} d_i \ , \qquad \text{t=1,...,T}$$

$$0 \leq z_t \leq \delta_t \sum_{i=1}^{T} d_i \ , \quad \text{t=1,...,T}$$

$$\delta_t \in \{0,1\} \ , \quad \text{t=1,...,T}$$

See [5] for the detailed mathematical derivation of this optimization problem.

[5] J. Muckstadt and A. Sapra. Principles of Inventory Management. Springer, 2010. Sections 4.1 and 4.2.1. Available at http://www.springer.com/us/book/9780387244921

Implementation of optimization problem in Pyomo

Implementation of optimization problem (11) is very straightforward and does not require any additional transformations. The optimization problem (11) is linear and hence can be solved efficiently using fast mixed integer programming solvers. Because of the legal constraints, our solution does not contain any such solvers preinstalled. As a temporary workaround, optimization problem (11) can be solved using pre-installed Bonmin solver. In section **How To Integrate New Solvers** we describe how to plug into our solution two fast mixed integer programming solvers, a free solver MIPCL and a commercial solver Gurobi.

Capacitated Dynamic Lot Sizing policy

Definition

Capacitated dynamic lot sizing problem is an extension of dynamic lot sizing policy where the total volume of the items delivered by supplier in a single time interval is limited. Unlike dynamic lot sizing policy, capacitated dynamic lot sizing policy manages a set of products that are delivered by the same supplier. Capacitated dynamic lot sizing policy can be used when supplier can allocate only a limited number of trucks for delivering new items.

Optimization problem for generating the policy

We use the similar notation to dynamic lot sizing policy:

- *N* – number of products managed by the policy
- h_i – storage cost of the i-th product in a time period
- K_i – cost of a single shipping of the i-th product
- T – number of time periods
- d_{ij} – predicted deterministic demand of the j-th product during the i-th time period
- $\delta_{ij} \in \{0,1\}$ – indicator if there will be an order of the j-th product at the beginning of the i-th time period
- z_{ij} – order size of j-th product at the i-th time period. $z_i > 0$ if δ_i=1, 0 otherwise.

The values of δ_{ij} and z_{ij} minimize shipping and storage costs and are found by solving optimization problem. The total shipping cost over the planning time period is

$$\sum_{t=1}^{T}\sum_{i=1}^{N} K_j \delta_{tj}$$

and the total storage cost over the same time period is

$$\sum_{t=1}^{T}\sum_{j=1}^{N} h_j \sum_{i=1}^{t}\left(z_{ij} - d_{ij}\right) .$$

The final optimization problem of dynamic lot sizing policy is

$$\min_{\delta_{tj}, z_{ij}} \quad \sum_{t=1}^{T}\sum_{j=1}^{N}\left\{K_j\delta_{tj} + h_i \sum_{i=1}^{t}\left(z_{ij} - d_{ij}\right)\right\} \qquad (12)$$

$$\text{s.t.} \quad \sum_{i=1}^{t} z_{ij} \geq \sum_{i=1}^{t} d_{ij} , \quad t=1,\ldots,T, \quad j=1,\ldots,N$$

$$0 \leq z_{tj} \leq \delta_{tj}\sum_{i=1}^{T} d_{ij} , \quad t=1,\ldots,T, \quad j=1,\ldots,N$$

$$\sum_{j=1}^{N} z_{tj} \leq C , \quad t=1,\ldots,T$$

$$\delta_{tj} \in \{0,1\} , \quad t=1,\ldots,T, \quad j=1,\ldots,N$$

See Section 2.1 of [6] for the derivation of optimization problem (12).

[6] A. Drexl and A. Kimms. Lot sizing and scheduling – survey and extensions. European Journal of Operational Research, 99:221-235, 1997. Available at http://homes.ieu.edu.tr/~aornek/ISE421_drexl_kimms.pdf

Implementation of optimization problem in Pyomo

Implementation of capacitated dynamic lot sizing policy follows the same lines as the one of dynamic lot sizing policy.

Newsvendor policy

Definition

Newsvendor policy manages products that have the same expiration date and short life time. Examples of such products are newpapers, dairy and produce products. Newsvendor policy assumes that managed products expire after a fixed time period (e.g. 1 day) and will be disposed. Also, newsvendor policy assumes that we can quantify the cost of missed sale opportunity, when there is a demand for an item but no items are available in the inventory. Newsvendor policy finds an optimal order size for the next time period. The order size depends on predicted demand, disposal costs for unsold items and the cost of missed sales and the purchase budget. Our implementation of newsvendor policy manages multiple products. See [6] for a description of simpler version of newsvendor policy that manages a single product.

Newsvendor policy has the following assumptions:

- products expire at the same day
- the lead time is very short and is negligible
- there is a deterministic forecast of demand for each managed product
- disposal cost and the cost of missing sale of each managed product are known
- purchase costs of all managed products are known
- a total purchase budget is known

[6] J. Muckstadt and A. Sapra. Principles of Inventory Management. Springer, 2010. Section 5. Available at http://www.springer.com/us/book/9780387244921

Optimization problem for generating the policy

We use the following notation:

V – total purchase budget
c_i – purchase cost of a single item of the i-th product
h_i – disposal cost of a single item of the i-th product
b_i – cost of missing sale of a single item of the i-th product
d_i – predicted deterministic demand of the i-th product for the next time period. Newsvendor policy transforms predicted deterministic demand into predicted probabilistic demand $p_i(x)$ with Poisson distribution and mean value d_i, $p_i(x) = e^{-d_i} d_i^x / x!$.
s_i – order size of the i-th product.

The optimal order sizes si are found by minimizing the combined disposal costs and the costs of missing sales, subject to the purchase budget. The final optimization problem of newsvendor policy is

$$\begin{aligned} \min_{s_1,\cdots,s_n} \ & \sum_{i=1}^{N} \mathrm{h}_i(s_i - d_i) + (\mathrm{h}_i + b_i) \sum_{x>s_i} (x - s_i) p_i(x) \\ \text{s.t.} \ & \sum_{i=1}^{n} c_i s_i \le V, \\ & s_i \text{ is nonnegative integer}, \ \ i = 1,\ 2, \cdots,\ N \end{aligned} \tag{13}$$

See [6] for a detailed mathematical derivation of this optimization problem.

Implementation

We use the same technique as in (*s,Q*) policy to replace the right infinite sum in (13) with finite one and to eliminate the optimized variables from the boundaries of summation:

$$\sum_{x>s_i}(x-s_i)p_i(x) \approx \sum_{x=0}^{V/c_i} (x-s_i)p_i(x)\frac{1}{1+\exp(-1\ \cdot(x-s_i))}\ .$$

The final optimization problem that is implemented using Pyomo library is:

$$\begin{aligned}&\min_{s_1,\cdots,s_n} \sum_{i=1}^{N} \mathrm{h}_i(s_i-d_i)+(\mathrm{h}_i+b_i)\sum_{x=0}^{V/c_i}(x-s_i)p_i(x)\frac{1}{1+\exp(-10\cdot(x-s_i))}\\ &\text{s.t. } \textstyle\sum_{i=1}^{n} c_i s_i \leq V,\\ &\qquad s_i \text{ is nonnegative integer, } \ i=1,\ 2,\cdots,\ N\end{aligned}$$

This is mixed integer non-linear optimization problem. We use Bonmin solver to solve it.