Impacts of Futures Markets Speculation and Rail Transportation Networks on Commodity Basis Behavior

Joshua D. Woodard, Cornell University (jdw277@cornell.edu)
Tridib Dutta, Cornell University
Lin Xue, Cornell University

Selected Paper prepared for presentation at the 2016 Agricultural & Applied Economics Association Annual Meeting, Boston, Massachusetts, July 31-August 2

 I would like to thank the Ag-Analytics.Org *team members for their assistance and support. Thanks also to USDA Chief Economist Robert Johansson and Peter Feather of the USDA Office of the Chief Economist for their insight and support. Parts of this work were supported by USDA Cooperative Agreement "Leveraging Data Integration Systems to Support Agricultural Policy Research: An Application to Rail & Transport Network Impacts on Agricultural Commodities".*

Impacts of Futures Markets Speculation and Rail Transportation Networks on Commodity Basis Behavior

Interactions between rail and transportation networks on commodity price behavior and grain flows remains an important issue in the agricultural sector, from both an industry and policy perspective (Casavant, 2015). Market access, network effects, and local conditions all play an important role in determining land-use allocation, trade, and price behavior in agricultural markets. While a large literature exists evaluating basis behavior and convergence from a price analysis perspective on local or partial scales, much less has been done on evaluating the impacts of rail and transport networks on basis behavior on larger scales, or incorporating hedging or speculative activities in related commodity future markets.

There has been much recent interest in evaluating the impacts of market access on agricultural valuations from a historical perspective (see e.g., Donaldson and Hornbeck, *forthcoming*), agricultural transport investments (Casavant, 2015), and impacts of rail infrastructure on economic activity in general (Donaldson, 2015). Nevertheless, incorporating rail network effects in such models remains challenging, and thorough investigations of the impact of rail infrastructure and related regulatory issues on spatial price basis behavior remain largely unexplored in holistic contexts.

This study briefly explores the determinants of commodity price basis and basis convergence with a particular focus on the influence of futures market speculation in conjunction with rail rates and transportation networks. This preliminary analysis sets out to replicate and explore some earlier approaches in the literature, incorporates for the first time to our knowledge futures markets position information, and also provides thoughts on future extensions. The *Data Appendix* to this paper also focuses on automation of data sourcing to enable real time analysis of these types of disparate market information and models relying on the Ag-Analytics.Org open data platform (Woodard, 2016a, Woodard, 2016b).

Data and Methods

Data were collected from a variety of disparate sources for this analysis. The main source of rail transportation data were obtained from various unstructured spreadsheets which are updated by USDA-AMS weekly. See the *Data Appendix* to this paper for information on the data automation routines; the data are also available for flexible and scalable querying via web-based API's at Ag-Analytics.Org. The rail tariff data are collected from Grain Transportation Cost Index report, while railcar secondary market data are obtained collected from the Weekly railcar bids/offers for the secondary non-shuttle and shuttle railcar Market data. We consider only Shuttle car bids for this analysis. For rail tariff data, we only consider the Monthly rail tariff including fuel surcharge. For the shuttle car bids data, we averaged over the 12 months bid values for a particular ending week.

Total rail car data were collected from the *Analysis of Association of American Railroads (AAR), Weekly Railroad Traffic Report*. Train Speed data were collected from *AAR's 'Railroad Performance Measure'* table. Traffic Volume data we also collected from the *Weekly Railroad Traffic* as total carloads plus Intermodal units. Data were also collected from *Surface Transportation Board's (STB) Freight Commodity Statistics* database. A report for each railroad company is published separately, and for each company, the dataset contains information for all the commodities transported. We focused on Corn for this analysis. Grain stocks values are obtained from the *National Agricultural Statistical Service*. The *Commodity Futures Trading Commission Commitment of Traders* (COT) dataset provides weekly observations from April 18th, 1995, to the present and provides information on positions of hedging (commercial) and non-hedging (non-commercial) entities on publicly traded futures exchanges (in this case the *Chicago Mercantile Exchange*), published weekly, for reportable traders. Position data are published for futures, as well as futures and options combined. Option open interest and traders' option positions are computed on a futures-equivalent basis using delta factors supplied by the exchanges. Long call and short put open interest is converted to long futures-equivalent open interest, and vice-versa in the combined report. Data are also reconciled for non-commercial spreaders, and investigated for non-reportable traders.

Model

Following OCE (2015) and Wilson and Dahl (2011), *origin bases* in the primary Midwest states are modeled as a function of *destination basis*, and other factors; the *destination bases* consolidated into two groups: Pacific North West (*PNW*) and Gulf of Mexico (*GOM*). PNW consists of Oregon, Washington and California, while GOM consists of Texas and Louisiana. For these two groups, we considered the average monthly basis in these states, defined as the spot price minus the nearby futures price. We also included variables for *Outstanding Sales*, *Tariff* (Monthly rail tariff including Fuel surcharge), and futures speculation measures from the *COT* database. *Outstanding Sales* are weekly export sales contracts of commodities at U.S. ports that have not been shipped at a given time, and *Tariff* is the monthly rail tariff for shipments of commodities including fuel surcharge. From the *CFTC COT* database, we evaluate percent of open interest from non-commercial longs (*OI_Noncommercial_Long_All*), non-reportable longs (*OI_Nonreportable_Long_Other*), the change in non-commercial long positions (*ChangeinNoncommercial_Long_All*), as well as the net long concentration ratio as published by the CFTC (*Concentration_NetLT4TDR_Long_All*). We considered weekly data from June, 2010 until middle of November 2015 (*N*= 1185).

Results

Several models were investigated for robustness, including various setups for fixed effects on the *Origin Basis* states (Minnesota, South Dakota, Nebraska, Iowa, and Illinois). Several combinations were run to investigate the relative orthogonality of the factors under consideration and their sensitivity to alternative specifications in an intentionally terse exposition.

Table 1.1 presents regression results with Origin State fixed effects and differential effect variables for *PNW* and *GOM* regions. Consistent with earlier studies, the PNW and GOM basis variables are positive and significant, reflecting spatial arbitrage relationships in basis; *PNW* has a larger marginal effect than *GOM*, with magnitudes ranging from 0.59-0.61, and 0.14-0.15, respectively. Long speculative (non-commercial) positions and changes in long speculative position in the futures market are negatively and statistically significantly related to basis (i.e., spot - futures), indicating that greater long speculative pressure has a spatially persistent upward impact on futures prices relative to the spatial complex of spot prices. This is true for *OI_Noncommercial_Long_All*, *ChangeinNoncommercial_Long_All,* as well as for the concentration ratio measure, *Concentration_GrossLT4TDR_Long_All.* Note that futures are only deliverable against a single set of locations in the origin states during any given period, while spot prices vary across origin as well as destination locations. On the other hand, non-reportable long position are positively related to *origin basis.* The *R-Sq* is fairly high in all models (approx. 0.89), with the vast majority of variance explained by the *destination basis* measures.

Table 1.1 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures					
	M1	**M2**	**M3**	**M4**	**M5**
PNW	0.610095 ***	0.595254 ***	0.600186 ***	0.611352 ***	0.610744 ***
GOM	0.159599 ***	0.149578 ***		0.156886 ***	0.157536 ***
OI_Noncommercial_Long_All		-0.004935 ***			
OI_Nonreportable_Long_Other			0.004276 ***		
ChangeinNoncommercial_Long_All				-0.000001 **	
Concentration_GrossLT4TDR_Long_All					-0.002502 **
Fixed Effect Origin State	Yes	Yes	Yes	Yes	Yes
N	2770	2770	2770	2770	2770
Adj.R^2	0.8966	0.9008	0.8976	0.8970	0.8968
Sigma^2	0.0204	0.0196	0.0202	0.0203	0.0204

*Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05*

Table 1.2 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures

	M6	M7	M8	M9	M10
PNW	0.610090 ***	0.568022 ***			
GOM	0.158685 ***	0.197077 ***			
*OI_Noncommercial_Long_All_*			-0.028956 ***		
*OI_Nonreportable_Long_Other_*				0.016625 ***	
*ChangeinNoncommercial_Long_All_*					-0.000001 *
*Concentration_GrossLT4TDR_Long_All_*					
*Concentration_NetLT4TDR_Long_All_*	-0.002112 *				
Outstanding Sales		-0.010213 ***			
Fixed Effect Origin State	Yes	Yes	Yes	Yes	Yes
N	2770	2770	2770	2770	2770
Adj.R^2	0.8968	0.9039	0.2144	0.0594	0.0441
Sigma^2	0.0204	0.0190	0.1550	0.1856	0.1886

*Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05*

Turning to table 1.2, which includes *Outstanding Sales*, the results for *PNW* and *GOM* basis are quite consistent, as are the estimates for the futures speculation measures. M8-M10 also evaluate model specification stability to dropping *PNW* and *GOM*, with the result that the speculative measures are significant, although some of the effect is attenuated to those remaining variables in their absence as would be expected, and the R-Sq drops significantly as expected to about 0.21 in *M8*. *Outstanding Sales* are significantly negatively related to origin basis, which is presumably due to spot prices falling in destination locations as outstanding sales in those locations increase, and inventories build.

Inspecting tables 1.2-1.8, *PNW* has a consistently larger marginal effect than *GOM* basis, as in Table 1.1. All other results are also robust and stable with regard to the speculative measures, regardless of whether fixed effects are employed or not for the *Origin States*. Table 1.6 includes the *Tariff* measure, which as expected is also negative, indicating that the imposition of rail tariffs places downward pressure on local spot prices relative to exchange traded futures prices.

Table 1.3 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures

	M11	M12	M13	M14	M15
PNW			0.562762 ***	0.567930 ***	0.569342 ***
GOM			0.184835 ***	0.197242 ***	0.194911 ***
OI_Noncommercial_Long_All			-0.003595 ***		
OI_Nonreportable_Long_Other				0.000093	
ChangeinNoncommercial_Long_All					-0.000001 *
Concentration_GrossLT4TDR_Long_All	-0.008257 **				
Concentration_NetLT4TDR_Long_All		-0.008006 **			
Outstanding Sales			-0.008866 ***	-0.010183 ***	-0.010082 ***
Fixed Effect Origin State	Yes	Yes	Yes	Yes	Yes
N	2770	2770	2770	2770	2770
Adj.R^2	0.0457	0.0460	0.9060	0.9039	0.9040
Sigma^2	0.1883	0.1882	0.0185	0.0190	0.0189

*Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05*

Table 1.4 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures

	M16	M17	M18	M19	M20
PNW	0.567138 ***	0.566029 ***			
GOM	0.195246 ***	0.197149 ***			
*x_ofOI_Noncommercial_Long_All_*			-0.023975 ***		
*x_ofOI_Nonreportable_Long_Other_*				-0.0031	
*ChangeinNoncommercial_Long_All_*					-0.000001
*Concentration_GrossLT4TDR_Long_All_*	-0.004409 ***				
*Concentration_NetLT4TDR_Long_All_*		-0.003916 ***			
Outstanding Sales	-0.010705 ***	-0.010695 ***	-0.026307 ***	-0.038259 ***	-0.037019 ***
Fixed Effect Origin State	Yes	Yes	Yes	Yes	Yes
N	2770	2770	2770	2770	2770
Adj.R^2	0.9047	0.9046	0.2685	0.1613	0.1611
Sigma^2	0.0188	0.0188	0.1443	0.1655	0.1655

*Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05*

Table 1.5 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures

	M21	M22
*Concentration_GrossLT4TDR_Long_All_*	-0.015846 ***	
*Concentration_NetLT4TDR_Long_All_*		-0.014410 ***
Outstanding Sales	-0.038733 ***	-0.038560 ***
Fixed Effect Origin State	Yes	Yes
N	2770	2770
Adj.R^2	0.1717	0.1713
Sigma^2	0.1634	0.1635

*Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05*

Table 1.6 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures

	M23	M24	M25	M26
PNW	0.678514 ***	0.680294 ***	0.713883 ***	0.714775 ***
GOM	0.106235 ***	0.112449 ***	0.075676 ***	0.080138 ***
Outstanding Sales	-0.007302 ***	-0.007207 ***		
Tariff	-0.370603 ***	-0.503861 ***	-0.415997 ***	-0.505544 ***
Fixed Effect Origin State	Yes	No	Yes	No
N	1185	1185	1185	1185
Adj.R^2	0.9296	0.9279	0.9267	0.9250
Sigma^2	0.0215	0.0220	0.0224	0.0229

*Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05*

Table 1.7 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures

	M27	M28
PNW	0.568022 ***	0.568022 ***
GOM	0.197077 ***	0.197077 ***
Outstanding Sales	-0.010213 ***	-0.010213 ***
Fixed Effect Origin State	Yes	No
N	1185	1185
Adj.R^2	0.9288	0.8840
Sigma^2	0.0217	0.0354

*Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05*

Table 1.8 - Regression Results of Origin Basis on Destination Basis, Rail Transport and Futures Speculation Measures

	M29	**M30**	**M31**	**M32**
*OI_Noncommercial_Long_All_*	-0.057034 ***	-0.070915 ***	-0.075892 ***	-0.099184 ***
Outstanding Sales				
Tariff (not Tcost)	3.679642 ***	-0.269846 ***	3.111209 ***	-1.106460 *
Interaction(OI_Noncommercial_Long_All_ and Tariff)			0.017854	0.026822
FixedEffect Origin State	Yes	No	Yes	No
N	1185	1185	1185	1185
Adj R^2	0.4453	0.2785	0.4457	0.2801
Sigma^2	0.1694	0.2203	0.1692	0.2198

*Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05*

Conclusions

While a handful of studies exist exploring basis behavior, and exploring impacts of trading behavior on futures prices, we are unaware of any study that explicitly takes into account futures positions of market participants in conjunction with rail transpiration effects in a nationwide model such as this. As highlighted in a recent study by the United States Department of Agriculture Office of the Chief Economist and the Agricultural Marketing Service (OCE, 2015), interactions between rail & transportation networks on commodity price behavior and grain flows remains an important issue in the agricultural sector from both an industry and policy perspective. Market access, network effects, and local conditions all play an important role in determining land-use allocation, trade, and price behavior in agricultural markets.

The findings of this brief study corroborate those of earlier studies as it regards frictional impacts of rail network costs on origin bases. Additionally, this study finds that in addition to rail transport effects, that long speculative pressure in the futures market is contemporaneously related to widening basis.

Future studies could investigate basis behavior in such markets in spatially explicit frameworks, and explore inclusion of ethanol demand. Additionally, despite that most grain is produced in the Central Midwest (what we and other designate as "origin" states) and shipped or consumed predominantly elsewhere (through "destination" states) we would note that these designations are somewhat *ad hoc,* and thus future explorations could instead employ more formal and spatially explicit econometric approaches to investigating such network effects in these markets.

References

Casavant, K.L, "Agricultural Grain Transportation: Are We Underinvesting and Why?" *Choices*, 2015.

Donaldson , D. and Hornbeck, R., "Railroads and American Economic Growth: A Market Access Approach," *Quarterly Journal of Economics*, forthcoming.

Donaldson , D., "Railroads of the Raj: Estimating the Impact of Transportation Infrastructure," *American Economic Review*, 2015.

National Research Council (NRC), 2012. Computing Research for Sustainability. Washington, DC: National Academies Press. 155 pp.

OFFICE OF MANAGEMENT AND BUDGET (OMB), "MEMORANDUM FOR THE HEADS OF EXECUTIVE DEPARTMENTS AND AGENCIES: Guidance for Providing and Using Administrative Data for Statistical Purposes", February 14, 2014

Office of the Chief Economist (OCE) - United States Department of Agriculture, "Rail Service Challenges in the Upper Midwest: Implications for Agricultural Sectors – Preliminary Analysis of the 2013 – 2014 Situation,, January 2015.

Office of Science and Technology Policy (OSTP), 2013. Increasing Access to the Results of Federally Funded Scientific Research. OSTP Memorandum for the Heads of Executive Departments and Agencies, February 22, 2013. Washington, DC.

President’s Council of Advisors on Science and Technology (PCAST), 2011. Sustaining Environmental Capital: Protecting Society and the Economy. Washington, D.C.

Wilson, William W. and Bruce Dahl (2010) “Grain Pricing and Transportation: Dynamics and Changes in Markets,” Agribusiness and Applied Economics Report No. 674, North Dakota State University (December)

Wilson, William W. and Bruce Dahl (2011), “Grain Pricing and Transportation: Dynamics and Changes in Markets” Agribusiness Journal, 27 (4) 420-434.

United States Department of Agriculture-Agricultural Marketing Service, *Grain Transportation Report Datasets* (2016), Unstructured source data are available online at https://www.ams.usda.gov/services/transportation-analysis/gtr-datasets

Woodard, J.D., "A Spatial Supply and Demand Analysis of United States Dairy Policy", *Working Paper*, Cornell University, 2015

Woodard J.D. (2016), Data Science and Management for Large Scale Empirical Applications in Agricultural and Applied Economics Research,” *Applied Economics Perspectives and Policy*

Woodard J.D. (2016), “Big Data and Ag-Analytics: An Open Source, Open Data Platform for Agricultural & Environmental Finance, Insurance, and Risk,” *Agricultural Finance Review*.

Data Appendix

This appendix highlights some key details of the data collection procedures employed and online resources developed in support of this study, including technical details of extraction, transform, load procedures (ETL) for automating data sourcing of unstructured data resources used in the pilot analysis. Difficulties we encountered during the process were recorded as well as suggestions for future opportunities for coordination improvements between the Federal Government and the research community in the realm of agricultural analytics.

Recent decades have seen an explosion in modeling capabilities and frameworks--particularly at the intersections of markets, policy, and spatially dependent enviro-economic systems--and software to perform such specific modeling tasks. Marrying of these worlds, however, remains seriously lacking. This is of profound importance given that many of the new modeling frameworks in spatial econometrics and economic network modeling require integrating and structuring large and complicated datasets consistent with these approaches, which is a massive task in and of itself. In fact, separately, fields of data science have emerged in several disciplines focusing on the latter alone (in which economics has arguably been a laggard relative to other sciences such as physics, computer science, genetics, and meteorology). In the course of pursuing this specific research, we also build off of successful approaches developed to-date for enabling automated and scalable data warehousing approaches in order to enable these investigations. These efforts are of critical importance for not only establishing proof of concept generally, but also motivating adoption of such research in a form that does not exist to date.

Importantly, unlike other systems to date, on the Ag-Analytics.Org platform, these data are stored in modern databases which can be queried and integrated flexibly and processed at user desired scales, with industrial and flexible API's and tools to most broadly enable analytics to the deepest extant community of users. To move the effort forward, more funding and emphasis should be placed on these robust cloud based approaches to allow for wide use, adoption, and further development, and to fully leverage other database and data management techniques, efforts, and platforms. To that end, we have obtained a generous grant from Microsoft for server space on their Azure platform, and also continue to move this exploratory effort forward on limited internal funds. A partnership between OCE and Cornell to further explore and develop within the context of Agency priority research questions under this study has also been of great value to the public, government agencies, and the associated research community to this end.

Communicating Proof of Concept and Developing Use Cases and Tools

While many advances have been made and successes realized to date in terms of fundamental technical and analytical challenges, a critically important and logical next step for advancements in this field will rely on meaningful interaction with interested agency stakeholders and leadership. In addition, meaningful articulation and demonstration of the business case for these innovations in practical and easily understood contexts is of great value. While it is understood that this working manuscript is a verbose working effort toward that, our hope is that inclusion and development of such resources will begin to spur interest within the community as the value and potential applications of such community resources. To this end, we have engaged the Office of the Chief Economist of the USDA in identifying and generating such expositions and cases. This *Data Appendix* also serves as a tutorial based on the pilot analysis in the paper highlighting the use of the data system, with the ancillary benefit that the research will be replicable in a manner that virtually no economic studies are to date in terms of enabling a auto-updatable and on-demand data sourcing.

An overarching purpose of this project is to explore leveraging data integration systems to support agricultural policy research in cooperation with The Office of the Chief Economist (OCE) of the United States Department of Agriculture (USDA). By recreating the analysis of Rail Service Challenges in the Upper Midwest[1] using a centralized and automated database, well documented ETL (Extract, Transform and Load) scripts, data modeling process and Metadata, the purpose of this effort is to further explore cooperation between the USDA OCE in order to set an example for future data integration projects leveraging community based open source/open data platforms such as Ag-Analytics.Org, where all of the data scripting routines, as well as a live-automated data warehouse are available for researchers to freely access.

Much of the data researchers routinely use in agricultural and environmental finance and related fields are often--strictly speaking--publicly available; however the form in which they are distributed leads to great inefficiencies in data sourcing and processing which can be greatly improved. This assessment has been widely supported even by the government for some time (OSTP, 2013; OMB, 2014). The goal of the Ag-Analytics open data/open source platform is to help researchers centralize, share, coordinate, and contribute in such efforts. Development of systems for disseminating, documenting, and automating the processing of such data can lead to more transparency in research, better routes for validation, and a more robust research community, and better expenditure of public funds.

The purpose of the remainder of this document is to provide an overview of the technical processes involved in extracting, curating, and housing various unstructured data for a set of use cases. Please refer to *"Big Data and Ag-Analytics: An Open Source, Open Data Platform for Agricultural & Environmental Finance, Insurance, and Risk", Agricultural Finance Review (2016, forthcoming),* and *"Data Science and Management for Large Scale Empirical Applications in Agricultural and Applied Economics Research, Applied Economics Perspectives and Policy (2016, forthcoming)* for further detail on conceptual design.

Metadata

Initial Metadata are collected from source on at least two levels: Table Meta data contain generic information about the dataset, such as title, description, author, source, format, license, coverage, update frequency, last revision date. Each row is a record of the transformed dataset collected. Fields Metadata contained detailed information of each field (column) inside the dataset. Table A.1 below provides a draft synopsis of those collected for this study. We would note that these data are used in a wide variety of contexts, but as of yet, the only available API to access these data are on the Ag-Analytics.org platform.

[1] Rail Service Challenges in the Upper Midwest: Implications for Agricultural Sectors – Preliminary Analysis of the 2013 – 2014 Situation

Dataset	Description	Source	Update
ExportGrainTotals	Federal Grain Inspection Services Yearly Export Grain Totals (data available from 1983 to current year)	USDA-FGIS	annual
ExportSalesWeeklyData	Weekly Export Grain Sales Data	USDA	Various
Fertilizer	Fertilizer records (in process)	ERS	Various
FreightCommodityStatistics	Quarterly and annual data for the seven major freight railroads. The major US railroad data used are {BNSF, CXWT,GTC, UP,SOO,NS,KCS}	Surface Transportation Board	annual
GrainInspectionByPort	Weekly inspections of grain for export in the Pacific NorthWest, Mississippi Gulf, Texas Gulf, Great Lakes and The Atlantic region.	AMS-USDA	weekly
GrainTransportByMode	Amount (in 1000 Tons) of US grain moved by rail, barge, and truck from 1978 to 2013. The data is divided into export, domestic and total grain moved by the three modes of transport.	USDA	weekly
GrainTransportCostIndex	Weekly changes in truck, rail, barge, and ocean freight rates using diesel prices, nearby secondary rail market rates, Illinois barge rates, and ocean freight rates from U.S. Gulf and Pacific NorthWest to Japan as proxies.	AMS-USDA	weekly
NASSCrops	NASS Crops database	USDA NASS	daily
RailTraffic	Weekly U.S. rail traffic data of Carloads, Intermodal Units, and Total Traffic from March 2013 to present.	Association of American Railroads	weekly
SecondaryRailcarBids	Weekly railcar bids/offers for the secondary non-shuttle and shuttle railcar Market.	AMS-USDA	weekly
TrainSpeedByCompany	This dataset is downloaded from Railroad Performance Measures website, where six major North American freight railroads have voluntarily reported three weekly performance measures. The weekly data shows the train speed (miles per hour) for intermodal, manifest, coal unit, and grain unit.	Railroad Performance Measures website	annual
WaybillSamples	It is a stratified sample of carload waybills for all U.S. rail traffic submitted by those rail carriers terminating 4,500 or more revenue	Surface Transportation Board	annual

	carloads annually. (2005-2014)

Table 1. Summary of Railroad Database. A more detailed Table Meta data is attached separated as Excel file.

ETL Process Overview

In general, many ETL processes within our system involve four steps:

1. Identify the source of raw data for tables and charts in the paper[1]
2. Write Python scripts to clean, correct, compile raw data into two dimensional tabular format that's ready for MS SQL server.
3. Write SQL command to create table with correct data types and upload the processed data table to MS SQL database.
4. Write Python scripts to update the data table and set SQL server job to periodically run the scripts.

Due to the wide variety of data sources and forms, the raw data initially enter in a variety of formats and need to be assessed by qualified researchers for how to best store such data for cataloging. This is typically done with an eye toward scalability to generic applications. Flat formats such as Excel, CSV and well formatted XML or JSON are usually the easiest to process. Despite that many public datasets are haphazardly published as PDFs, they typically require the most ad hoc and unreliable conversion, as the resulting text usually loses its tabular format, and hence can be fairly difficult to parse. All agree that Agencies and data publication entities should avoid publishing (as a matter of unique record) data in this format more or less exclusively.

Most of our data sources are collections of multiple files. For example, FreightCommodityStatistics combines reports from seven major railroad companies, each recorded data in a different format. In some cases, such as with weather data, the number of files necessary to construct a usable dataset stretches into the thousands or tens of thousands, so basis scripting is necessary, and should be commoditized when possible. In some (but not most) cases, it is necessary with current technology to partition files. Yet, this is common even in very small datasets. Despite this obviousness, the practice of publishing data in this form is pervasive. While not difficult to overcome for the astute programmer with some time, the fact of the matter is that classic DBMS is a much preferred alternative usually, especially when such processing can be shared or centralized for ad hoc querying. For example, the *RailTraffic* table combines over 300 weekly reports for which it was necessary to write web-scrapping scripts to retrieve. A weekly update SQL job is set for it as a new report is posted every week. While feasible for every researcher to rewrite themselves (maybe), indeed many an analyst has either labored at great error to copy-paste ad-infinitum, re-script, or simply walk away from such data. To be sure, these data are widely underutilized relative to what they could be.

Application for collection of Rail Traffic Databases

The examples below present some very basic applications. Surely, the process of storing data in a cloud based open data platform for querying against a live DBMS is not new.

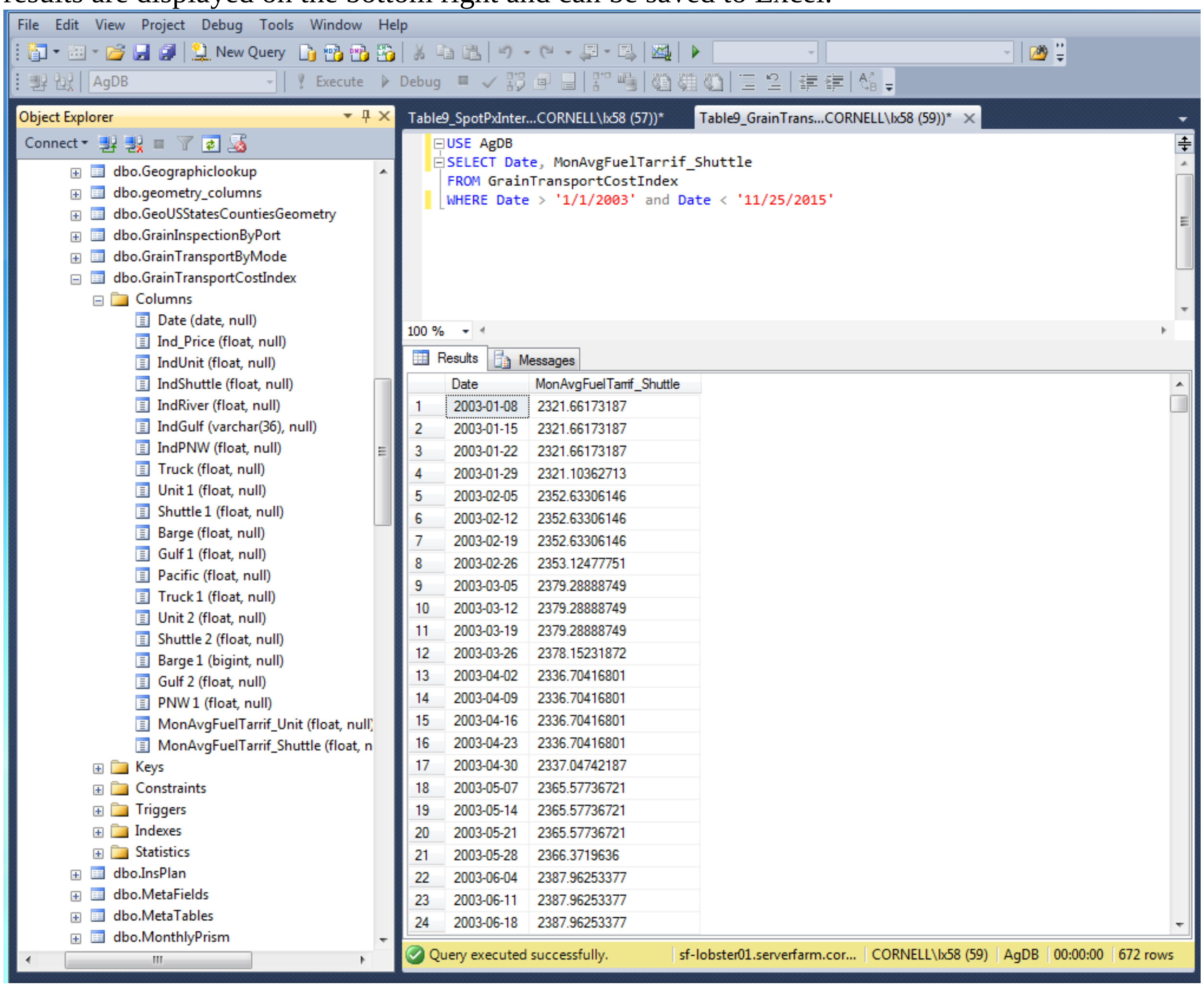
Query Example 1:

Query table GrainTransportCostIndex to get values for monthly average tariff including fuel surcharge for shuttle cars from the ag-analytics API (copy and paste the below into any web browser, or URL load program in any standard stat package):

https://ag-analytics.org/AgRiskManagement/api/dataservice?sql=

SELECT Date, MonAvgFuelTarrif_Shuttle FROM GrainTransportCostIndex WHERE Date > '1/1/2003' and Date < '11/25/2015'SELECT Date, MonAvgFuelTarrif_Shuttle
FROM GrainTransportCostIndex
WHERE Date > '1/1/2003' and Date < '11/25/2015'

Below is a screenshot of MS SQL Server Management Studio. Left column is a list of tables and their columns in our database. The user type in the above SQL query on top right section and results are displayed on the bottom right and can be saved to Excel.

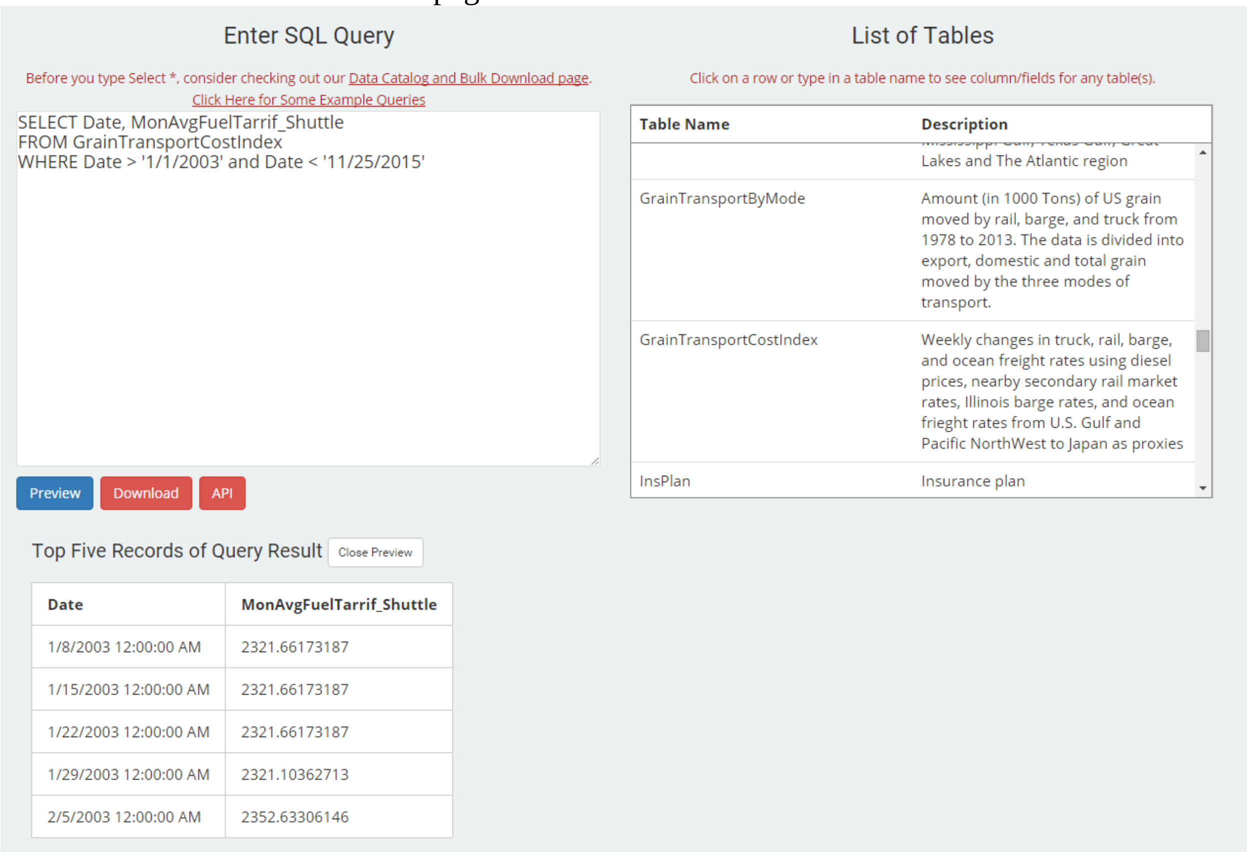


Below is a Matlab command to run the above SQL query and get data from database where you save the SQL query in filename.sql and sqlweb is an in house Matlab command written by Prof. Joshua Woodard. See https://ag-analytics.org/AgDBForum/topic12-call-web-api-from-matlab-using-sqlwebm.aspx

```
%MATLAB CODE-Ensure that SQLWEB.m function is in path

SQLSTRING1 = fileread('filename.sql');
[resultMatrix,FieldNames] = sqlweb(SQLSTRING1);
```

Though not everyone has access to MS SQL management studio, common users can type the above SQL query on our web interface and download the result:
https://www.ag-analytics.org/AgRiskManagement/ResAgDataQuery
Below is a screenshot of our web page.



Researchers can also access our database via API call. The URL for the above query is:

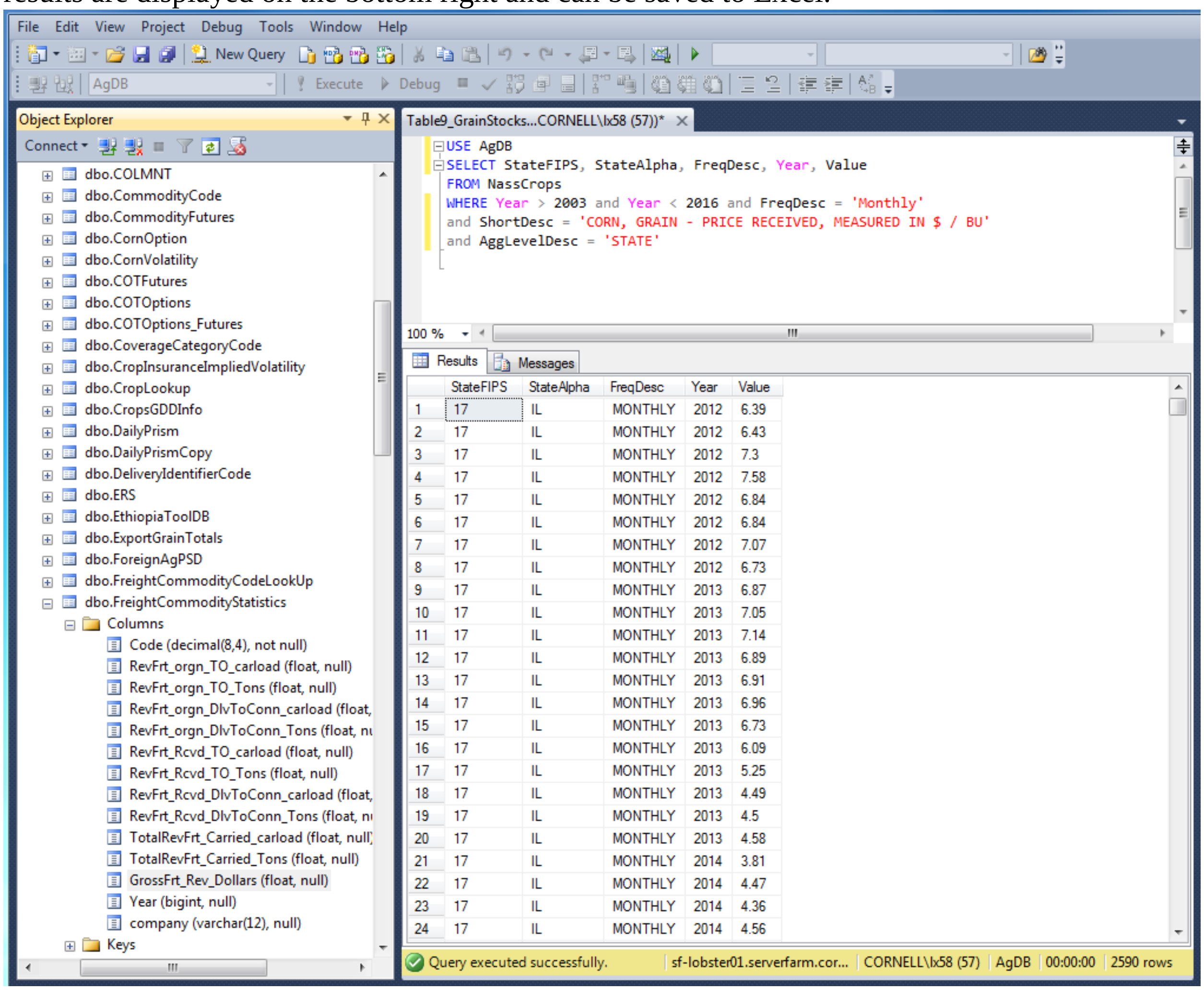
```
http://ag-analytics.org/AgRiskManagement/api/dataservice?sql=SELECT Date, MonAvgFuelTarrif_Shuttle FROM GrainTransportCostIndex WHERE Date > '1/1/2003' and Date < '11/25/2015'
```

Query Example 2:

Query table NASSCrops for monthly average corn price (measured in dollar per Bushel) from 2003 to 2016 in state of Illinois.

> http://ag-analytics.org/AgRiskManagement/api/dataservice?sql=SELECT StateFIPS, StateAlpha, FreqDesc, Year, Value
> FROM NassCrops
> WHERE Year > 2003 and Year < 2016 and FreqDesc = 'Monthly'
> and ShortDesc = 'CORN, GRAIN - PRICE RECEIVED, MEASURED IN $ / BU'
> and AggLevelDesc = 'STATE'

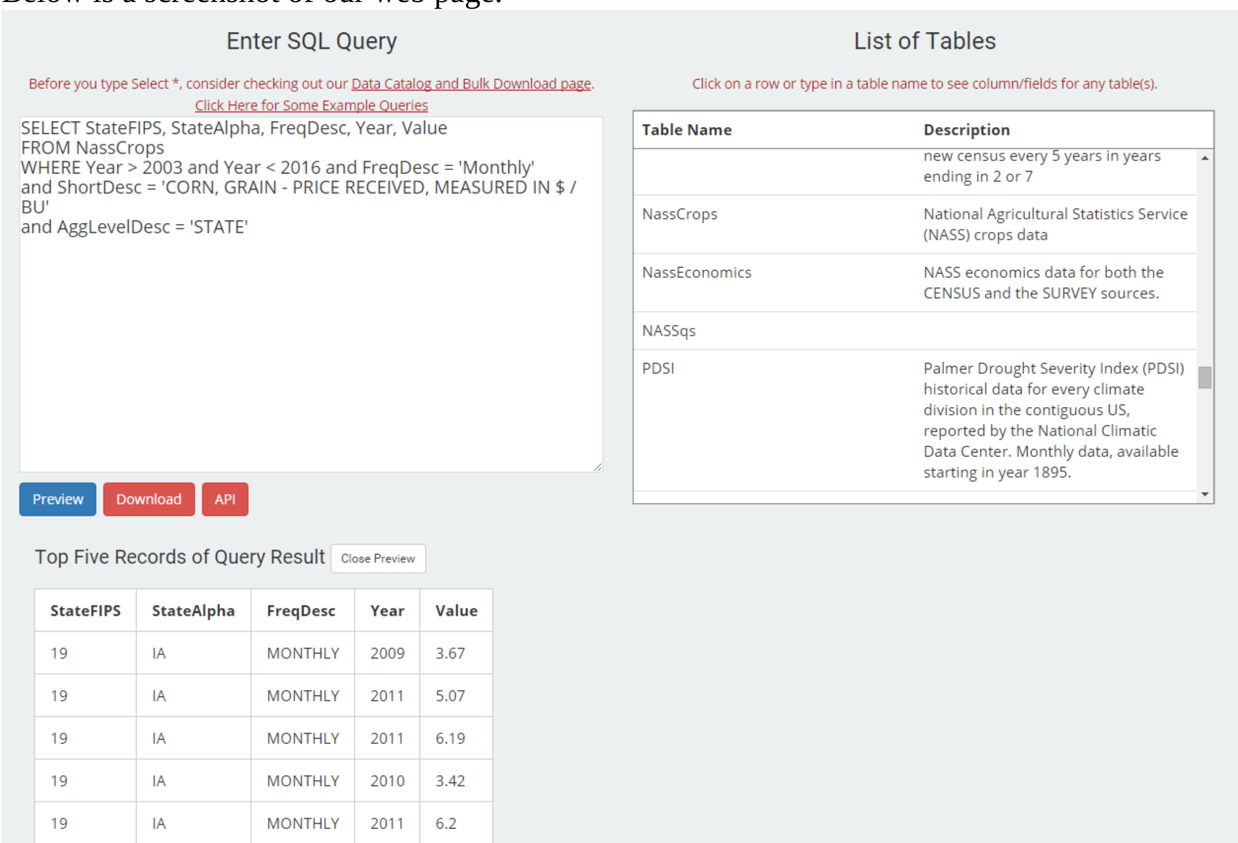
Below is a screenshot of MS SQL Server Management Studio. Left column is a list of tables and their columns in our database. The user type in the above SQL query on top right section and results are displayed on the bottom right and can be saved to Excel.



Below is a Matlab command to run the above SQL query and get data from database where you save the SQL query in filename.sql and sqlweb is an in house Matlab command written by Prof. Joshua Woodard and available at ag-analytics.org in the Forum and in API documentation examples. See https://ag-analytics.org/AgDBForum/topic12-call-web-api-from-matlab-using-sqlwebm.aspx

```
   %MATLAB CODE-Ensure that SQLWEB.m function is in path
SQLSTRING1 = fileread('filename.sql');
   [resultMatrix,FieldNames] = sqlweb(SQLSTRING1);
```

Though not everyone has access to MS SQL management studio, common users can type the above SQL query on our web interface and download the result:
https://www.ag-analytics.org/AgRiskManagement/ResAgDataQuery
Below is a screenshot of our web page.



Researchers can also access our database via API call. The URL for the above query is:

```
http://ag-analytics.org/AgRiskManagement/api/dataservice?sql=SELECT
StateFIPS, StateAlpha, FreqDesc, Year, Value FROM NassCrops WHERE Year
> 2003 and Year < 2016 and FreqDesc = 'Monthly'  and ShortDesc = 'CORN,
GRAIN - PRICE RECEIVED, MEASURED IN $ / BU'  and AggLevelDesc =
'STATE'
```

Technical Details by Dataset for this example study

ExportGrainTotals

The source data was a list of CSV files on this webpage from USDA-FGIS:
https://www.gipsa.usda.gov/fgis/exportgrain/

- Export Grain Inspection 2016 (last updated 3/7/2016 4:20:58 PM)
- Export Grain Inspection 2015 (last updated 2/16/2016 7:01:47 AM)
- Export Grain Inspection 2014 (last updated 5/26/2015 4:34:21 PM)
- Export Grain Inspection 2013 (last updated 6/20/2014 8:22:09 AM)
- Export Grain Inspection 2012 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2011 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2010 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2009 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2008 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2007 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2006 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2005 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2004 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2003 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2002 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2001 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2000 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1999 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1998 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1997 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1996 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1995 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1994 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1993 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1992 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1991 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1990 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1989 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1988 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1987 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1986 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1985 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1984 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 1983 (last updated 4/8/2013 12:00:00 AM)

Shown below is a small section of one CSV file.

Thursday	Serial No.	Type Serv	Cert Date	Grain	Pounds	Destination	Subl/Carrs
20160107	390469	IW	20160102	WHEAT	58876075	PHILIPPINES	30
20160107	390470	IW	20160103	WHEAT	24250600	PHILIPPINES	13
20160107	390471	IW	20160102	CORN	45158400	PERU	16
20160107	390472	IW	20160103	SOYBEANS	61392110	CHINA MAIN	17
20160107	390473	IW	20160103	SOYBEANS	77773850	CHINA MAIN	23
20160107	390474	IW	20160104	CORN	50320030	MEXICO	18
20160107	390475	IW	20160104	WHEAT	12125120	MEXICO	4
20160107	390481	I	20160102	CORN	24200000	MEXICO	110

The ETL process is as follows:
Step 1: Use YearlyExportGrainTotal.py to fetch links from USDA site, download all the csv files, append them one after another to form a larger csv and write into 11_processed.csv
Step 2: Use Upload_YearlyExportGrainTotal.py to create table in AgDB [dbo].[ExportGrainTotals] and bcp write to SQL server
Shown below is the python script for Step 1.

```
###########################
######Date: 01-25-2016
Copyright: Joshua D. Woodard, Ag-Analytics.org
Contributors: Lin Xue, Tridib Dutta, Josh Woodard, with assistance from Alex Muchocki, Anthony Perello, and Ag-Analytics team.
###########################
######Summary: This script fetch links from usda site, download the csv files, append them one after another to form a larger csv
############################

## packages needed
import urllib2
import re
import os
import pandas as pd
import ssl

## global variable
```

```python
workingDir = 'E:\\DatabaseFiles\\UpdateDBFiles\\ExportGrainTotals\\'
pathToFiles = 'https://www.gipsa.usda.gov/fgis/exportgrain/'
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE
## function to fetch the links to the required .csv files
def fetchLinks(pathToFiles):
    lines = urllib2.urlopen(pathToFiles, context = ctx).readlines()
    lines = ''.join(lines)
    links = re.findall('(\S+).csv', lines)
    #print links
    #create the links corresponding to each file and store it in a list
    #the line below: creats a string out of the list links
    #replace the unwanted "'" by re.sub() and then replace the 'href=' by the
pathToFiles and split() into a list
    csvLinks = re.sub('[\']','', ' '.join(links).replace('href=',pathToFiles)).split(' ')
    for i in range(len(csvLinks)):
        csvLinks[i] = csvLinks[i] +'.csv'
    return csvLinks

## function to download a file from given link
def downloadfiles(link):
    try:
        fileName = url.split('/')[5]
        #print fileName
##          testFile = urllib.FancyURLopener()
##          testFile.retrieve(url, workingDir + fileName)
##          #csvFile.write(testFile.read())
##          testFile.close()
        response = urllib2.urlopen(url, context=ctx)
        #open the file for writing
        fh= open(workingDir + fileName, "w")
        #read from request while writing to file
        fh.write(response.read())
        fh.close()
    except IOError as e:
        print e

## main()
## get the links (important variable, we will use it later as well)
reqLinks = fetchLinks(pathToFiles)
#read the files and dump those onto the working directory
for url in reqLinks:
    fileName = url.split('/')[5]
    downloadfiles(url)
```

```
###reqLinks[len(reqLinks)-1]
##reqLinks[::][0].split('/')[5]

##To build a single dataset, read in the files and stack one on top of the other
#First, build an empty dataframe
df = pd.DataFrame()
#cycle through the downloaded files and append one after the other
for x in reqLinks[::]:
   fileName = x.split('/')[5]
   DF = pd.read_csv(workingDir + fileName,low_memory=False)
   ##some of the cells in column 6 contain comma like " , ", remove the comma as
we are writing csv
   DF.iloc[:,6] = DF.iloc[:,6].str.replace(',',' ')
   df = df.append(DF, ignore_index=True)

## now that the dataset is build, dump it in the RailWay_database as
11_processed.csv
df.to_csv(workingDir + '11_processed.csv', index = False)
```

Set up SQL Server Agent Job to update monthly.
SQL Job Name: OCE_ExportGrainTotals
Step 1: C:\python27\python
E:\DatabaseFiles\UpdateDBFiles\ExportGrainTotals\YearlyExportGrainTotal.py
Step 2: C:\python27\python
E:\DatabaseFiles\UpdateDBFiles\ExportGrainTotals\Upload_YearlyExportGrainTotal.py
Schedules: Monthly on day 1

FreightCommodityStatistics ETL Procedure Example

For this example, we scrape Freight Commodity Statistics dataset from Surface Transportation Board's (STB) website (http://www.stb.dot.gov/econdata.nsf/FCStatistics?OpenView). These are historical datasets which are available from the year 2012 onwards. The earlier years have datasets in scanned pdf formats making them virtually useless and incapable of scraping or using. There are altogether seven railroad companies whose commodity statistics data are provided in the STB's website. They are Burlington Northern Santa Fe (BNSF), Union Pacific (UP), Grand Trunk Corporation (GTC), CSX Transportation (CXWT), Norfolk Southern (NS), Soo Line Railroad (SOO), and Kansas City Southern Careers (KCS).
For each of these companies, the following information are collected. There are two major categories: Revenue Freight Originating on respondent's Road and Revenue Freight Received from connecting carriers. Within each of these two categories, there are two sub-category: Terminating on line and Delivered to Connecting Carriers. Within each of these sub-categories, there are two sets of numbers that are given: Number of Carloads and Number of Tons (2000 LBS). The Total Revenue Freight Carried and Gross Freight Revenue Dollars are also provided. Each row of table contains the commodity the railway carried; full description and a corresponding numeric code is also provided. Most the companies refer to this numeric code to describe the commodity they carried.

The tables are not wholly consistent with each other, and contain numerous data entry errors. For each of the companies, we had to write a script to download the data (the excel file) and process that dataset into a usable format. Finally we combined those dataset into a single small data table before uploading to DBMS for querying via cloud platform. To identify which data belongs to which company and which year in the final table, we added two extra columns; one representing year of the statistics and the second represents the company whose statistics it is. The following python scripts need to be run in order to build the processed flat file from the available excel files on the web. Note: We have noticed that some of the URLs may have changed within the timeframe of our project. So if one or more script doesn't return anything, then please check the URL first.

```
##Step 1
Run FrightComodityStatistics_{BNSF, GTC, CXWT, NS, SOO, UP, KCS}.py
scripts
```

These scripts will download raw data to a temporary subfolder 'tempDirFor_06' and then uses this raw data to build the processed table for each company (NS etc.).
Step 1 above represents seven scripts, one for each company. Below is an example of a typical script.

```
### This script gets the data for the company BNSF
### The raw data is already there in the tempDir_06 subfolder

import pandas as pd
import urllib
import chardet
```

These modules were used in the script to scrape and process the data.

```

#recode the CODE column for consistancy
def recode(x):
   #calc len of x (this would be like number of rows in your file
   N=len(x)
   #loop thorough and do checks then recode
   currTree=0 #set current tree val at zero
   currTreeLen=1 #
   recodeV= [None] * N

   for i in range(N):
      #need to convert x[i] to integer first since codeing of 01 is not consistent, the convert backto string to strip lieading zeros
      y=int(x[i])
      v=str(y)
      chkTree=int(v[:currTreeLen])
      #check if equal to current tree num
      if chkTree==currTree:
         #if so recode by stripping of currTreeLen digits
```

```
        recodeV[i]=float(str(currTree)+'.'+v[currTreeLen:])
        #print str(currTree)+'.'+v[currTreeLen:]
      else:
        recodeV[i]=int(v)
        currTree=int(v)
        currTreeLen=len(v)

  return recodeV
```

The above code is needed to redo the code for each commodity. Due to the ambiguity present in the coding, we felt the need for recoding into a different format which is consistent and which can be used as a key in a database table.

```
## A boolean function checks to see if it x is an integer
def is_int(x):
  try:
    int(x)
    return True
  except:
    return False

## This function gets the row indices where there is not an integer
## ‘a’ is a pandas data frame
## colIndex is the column index which we want to test
def findBadIndices(a,colIndex):
  badIndices = []
  for i in range(len(a)):
    x = a.iloc[i,colIndex]
    if not is_int(x):
      badIndices.append(i)
  return badIndices

## boolean: number or not (could be int, float etc)
def is_number(x):
  try:
    float(x)
    return True
  except ValueError:
    return False
```

```
## Boolean: checks if a string is not emplty
def isNotBlank (myString):
  if myString and myString.strip():
    #myString is not None AND myString is not empty or blank
    return True
```

```
    #myString is None OR myString is empty or blank
    return False


## these will clean up the cells of each table (there are lots of errors)

## There are couple of datetime object burried deep inside the excel file
## this function checks if an entry is a pandas datetime object
def is_dateTimeObj(x):
    if isinstance(x, pd.datetime):
        return True
    else:
        return False
```

The code snippet above has several custom functions which will help us in the function below.

```
def clean_cell(some_string):
    if not pd.isnull(some_string):
        if is_dateTimeObj(some_string):
            return None
        if type(some_string) == unicode:
            ## convert to ascii
            some_string = some_string.encode('ascii','ignore')
        if isinstance(some_string, str):
            if chardet.detect(some_string)['encoding'] == 'ascii':
                if is_number(some_string):
                    if is_int(some_string):
                        some_string = int(some_string)
                        return some_string
                    else:
                        some_string = float(some_string)
                        return some_string
                else:
                    if len(some_string.split('/')) > 1:
                        some_string = None
                        return some_string
                    elif len(some_string.split('-')) > 1:
                        some_string = None
                        return some_string
                    else:
                        if isNotBlank(some_string):
                            some_string = ''.join(c for c in some_string if c.isdigit())
                            if isNotBlank(some_string):
                                return some_string
                            else:
                                return None
```

```
                else:
                    some_string = None
                    return some_string
            else:
                #chardet.detect(some_string)['encoding'] != 'ascii':
                some_string = None
        else:
            return some_string
    return some_string
```

The workhorse for our python script is the above function which cleans each cell with errors. The errors could be one of many. For example, a few of the entries were found to be a data-time object while many of the entries had special symbols '-' or ',' or white space in between digits of a number, making them read in as character and not as a numeric value. Some even had encoding problems which we needed to get rid off or if possible convert to ascii.

```
## The following function outputs a dictionary with data type groups for the columns of the dataFrame
def dataTypeOutput(dataFrame):
    g = dataFrame.columns.to_series().groupby(dataFrame.dtypes).groups
    return g
```

```
url = 'http://www.stb.dot.gov/econdata.nsf/27dead93525f6773852578aa004bc24d/4a7f061966ae7fcb85257dfd00572bbd/$FILE/BNSF.xlsx'
testFile = urllib.URLopener()
testFile.retrieve(url,'tempDirFor_06/06_BNSF_output_2014.xlsx')
```

The above line of codes retrieve the data file from the provided URL.

```
skip_rows = 9
df_2014 = pd.DataFrame()
for i in range(12):
    if i != 11:
        df = pd.read_excel('tempDirFor_06/06_BNSF_output_2014.xlsx', skiprows = skip_rows, header = None,sheetname=i)
        df_2014 = pd.concat([df_2014, df], ignore_index=True)
    else:
        df = pd.read_excel('tempDirFor_06/06_BNSF_output_2014.xlsx', skiprows = 10, skipfooter = 14 ,header = None, sheetname = i)
        df_2014 = pd.concat([df_2014,df], ignore_index = True)
```

After examining the raw data, we needed to set the parameters of the pandas' read_csv() function accordingly. Above code snippet does exactly that.

The first column (0th column) contains both the Code and their values(names)
For consistency, we get rid off the code names and keep only the code
For consistency, we split up the first (index 0) INTO TWO COLUMNS (containing the code and the name) as in below.

```
newDF = pd.DataFrame(df_2014.iloc[:,0].str.split(' ',1).tolist(), columns = ['0','1'])
```

Drop the first column and concatinate the newDF with it and then rename the columns
This line of code get rid off the code names

```
df_2014.drop(df_2014.columns[0],axis =1, inplace = True)
```

Concatenate newDF and df_2014

```
df_2014 = pd.concat([newDF[newDF.columns[0]], df_2014], axis = 1)
```

Remove any row with 0 columns having NaN

```
df_2014.drop(df_2014.index[findBadIndices(df_2014,0)], inplace = True)
```

Re-index the rows (to be consistent)

```
df_2014.index = list(range(len(df_2014)))
```

Rename the columns (to be consistant)

```
df_2014.columns = list(range(len(df_2014.columns)))
```

Recode the first column (using the recode() function).

```
df_2014.iloc[:,0] = recode(df_2014.iloc[:,0].values.tolist())

for i in range(1,12):
    df_2014.iloc[:,i] = df_2014.iloc[:,i].apply(clean_cell)
```

The other years, 2012 and 2013 are somewhat similar to 2014 in principle and we omit it from this report for the sake of clarity and brevity.

```
## add a 'Year' column to the dataframes
a = pd.Series([2014]*len(df_2014), dtype = 'int', name = 'Year')
df_2014 = pd.concat([df_2014,a], axis = 1, ignore_index=True)

## add a 'Year' column to the dataframes
a = pd.Series([2013]*len(df_2013), dtype = 'int', name = 'Year')
df_2013 = pd.concat([df_2013,a], axis = 1, ignore_index=True)

## add a 'Year' column to the dataframes
a = pd.Series([2012]*len(df_2012), dtype = 'int', name = 'Year')
df_2012 = pd.concat([df_2012,a], axis = 1, ignore_index=True)
```

```
#stack the different data frames according to the year
final_df = pd.concat([df_2012,df_2013,df_2014], ignore_index=True)
```

The above code snippet compiles the different years into a final table.
Once we process all the companies. We have seven datasets. We compile those seven into a single large table in Step 2 below.

```
##Step 2
Run FrightComodityStatistics_buildTable.py
```

Note that this will create the final flatfile (.csv) by combining the cleaned up and processed files from Step 1 for each of the companies mentioned above.

```
##Step 3
##to upload the table to a SQL server
Run FrightComodityStatistics_UPLOAD.py
```

Note 1. You must change the following information according to your need: sql CREATE TABLE statement, sqlserverinstance, database schema, working director, filename containing the table, table name etc.

Note 2. We ran into problems with 'bcp' command (for bulk upload to SQL server). In most of the cases we were able to resolve the issue by setting appropriate flag in the 'bcp' command for end-of-line (EOF) charachter (it could be either {CR}{LF} or {CR} or {LF}. Check yours by opening it with, say, Notepad++ and enabling hidden symbol viewing capabiity).
Here is a snapshot of the final dataset.

Code	Revenue Frieight Origin Terminating Online Carload	Revenue Frieight Received Terminating Online Carload	Total Revenue Frieight Carried Carload	Year	company
1	657042	18158	780980	2012	BNSF
1.1	613499	17513	733412	2012	BNSF
1.12	10710	2	10712	2012	BNSF
1.121	0	0	0	2012	BNSF

We encountered the following issues we ran into while processing the files for each company.

1. The url for some of the excel raw data file seems to have changed very recently and URLopener() function from the `urllib` module no longer works. We needed to replace `URLopener(`) by `FancyURLopener()` to make them retrieve the target raw data file.
2. Several of the excel files for companies including BNSF, GTC, etc. have multiple format error making them really hard to make a flat file (`.csv`) out of those file.

All the scripts in Step 1 contain a function called `clean_cell()` which takes care of most of those formating abnormalities buried deep inside those excel raw data files. For example, in the raw data file for the year 2012 from the company BNSF, most of the entered numbers had space in them (`2124 4` instead of 21244) making them read in as `string` in the database or python reader, resulting in obvious misinterpretation while performing analysis or any mathematical operations on these datasets. Couple of the cells seems to have accidentally encoded as a `datetime` object when they should be clearly numeric. Our `clean_cell()` function is able to handle all these problems. However, there might be unexpected formating error in the file which `clean_cell()` may not be able to process.

3. Another major issue we ran into is the encoding of different goods/Commodities that the railway companies transported. It seems that there is a industry-wide standard for encoding commodity goods the railway companies handle. They have assigned integer code to different goods. For example the number 1 (sometimes entered as 01, although not consistently) represents a broad category which is called 'Farm Products'. Under this category is 'Field Crops', which is assigned a value 11 (sometimes entered as 011, , although not consistently). Ironically, in the same document, the number 11 was assigned to 'Coal' which is a broad category representing coal based products such as Anthracite which was assigned a numeric value of 111. This we thought could create a lot of confusion especially if this coding is used as 'key' while downloading data from our database. To resolve the matter, we had to come up with our own recoding of these numeric values. We decided that it is fit that instead of using integer values, we would use float type numbers and 1 will be encoded as 1.000 while 11 (representing Field Crops, a sub category of Farm product) will be assigned the value 1.100 instead of 11 or 011. This way coal can be 11.000 and so on. We wrote a little function called `recode()` which can be found in each of the scripts mentioned in Step 1.

GrainInspectionByPort

This table contains data for the grain inspected and/or weighed for export by region and port region. The port regions are Pacific North West, Mississippi Gulf, Texas Gulf, Interior region, Greal Lakes region and the Atlantic region. The data shows insepction for Corn, Wheat and Soybean at these ports. The table contains data starting from 1/4/1996 to the present (as of the writing of this report).

```
Copyright: Joshua D. Woodard, Ag-Analytics.org
Contributors: Lin Xue, Tridib Dutta, Josh Woodard, with assistance from Alex
Muchocki, Anthony Perello, and Ag-Analytics team.
###########################
######Summary: This script download the GTRTable16.xlsx from the USDA
website:
###### http://www.ams.usda.gov/services/transportation-analysis/gtr-datasets,
###### read and process the table using pandas and save it to a CSV file.
############################
import os
import urllib
import pandas as pd
```

The above modules are used to scrape and process the file.

```
#global variables
workingDir = "E:\\DatabaseFiles\\UpdateDBFiles\\GrainInspectionByPort"
```

```
##retrieve the file from the link
url = 'http://www.ams.usda.gov/sites/default/files/media/GTRTable16.xlsx'
testFile = urllib.URLopener()
testFile.retrieve(url, workingDir + '\\12_output.xlsx')
```

The above code snippet retrieves the original raw data file (GTRTable16.xlsx) from the USDA website and saves it as 12_output.xlsx in our local folder.

```
## takes a datetime.datetime obj and converts into this specific format
def dateTimeToNormal(date):
   return date.strftime('%m/%d/%Y')
```

This is a utility function which will be used later while processing data.

The set of codes below read in the saved file (12_output.xlsx) into a pandas data frame and change the names of the columns to a more readable names.

```
usecols =
['Date','Wheat','Corn','Soybean','Wheat.1','Corn.1','Soybean.1','Wheat.2','Corn.2','S
oybean.2',
'Wheat.3','Corn.3','Soybean.3','Wheat.4','Corn.4','Soybean.4','Wheat.5','Corn.5','So
ybean.5']

##read in the excel file into a pandas data frame DF
DF = pd.read_excel('12_output.xlsx', sheetname = 'Data',skiprows =
20,skip_footer = 50, usecols = usecols)

##change the column names to the following
newColNames =
['Date','pac_Wheat','pac_Corn','pac_Soybean','MS_Wheat','MS_Corn','MS_Soybe
an','TX_Wheat','TX_Corn','TX_Soybean','GL_Wheat','GL_Corn','GL_Soybean','
Atl_Wheat','Atl_Corn','Atl_Soybean','Int_Wheat','Int_Corn','Int_Soybean']

##change the colnames
for i in range(len(newColNames)):
   DF.columns.values[i] = newColNames[i]
```

As is typical of these raw datasets, there were seveal formating errors in the saved .xlsx file. The functions below will find objects with unicode encoding among the datetime objects and return a cleaned datetime object if possible.

```
## There seems to be unicode objects buried in this column as seen above.
## Need to clean those and convert to datetime.datetime object
```

```
## First need to find the indices of those unicode values
def returnBadIndices(DF):
   bad_vals = []
   for i in range(DF.shape[0]):
      if type(DF.Date[i]) is unicode:
         bad_vals.append(i)
         #print i
   return bad_vals

bad_indices = returnBadIndices(DF)

## return cleaned datetime.datetime objects

def returnCleanedDateTimeObjs(bad_vals, DF):
   import re
   from datetime import datetime
   cleanedVals = []
   dateList = []
   for i in range(len(bad_vals)):
      a = DF.ix[bad_vals[i],'Date'].split('/')
      a[2] = re.sub('[^0-9]','', a[2])
      a = a[2]+'-'+a[0]+'-'+a[1]

      dta = datetime.strptime(a,'%Y-%m-%d') ## necessary to make it uniform
with the rest of the column vals
      cleanedVals.append(dta)
   return cleanedVals

## Fix the errors here
goodVals = returnCleanedDateTimeObjs(bad_indices,DF)
for i in range(len(bad_indices)):
   DF.ix[bad_indices[i],'Date'] = goodVals[i] ## assigned the cleanedup values to
the appropriate places
```

```
#convert to the required format using dateTimeToNormal() function
DF.Date = DF.Date.map(lambda x: dateTimeToNormal(x))
```

Once the date objects are cleaned and properly formated, we write it back to our local folder as '12_processed.csv' file for the next step, which is to upload it to SQL server.

```
##write to a .csv file in preparation for upload to the SQL database
DF.to_csv(workingDir +'\\12_processed.csv',index = False)
```

```
############ Remove the temporary file
os.remove(workingDir+'\\12_output.xlsx')
```

A snapshot of the data is presented below.

Date	pac_Wheat	pac_Soybean	MS_Wheat	MS_Corn	MS_Soybean	TX_Wheat	TX_Corn
1/4/1996	11540	3592	1827	26476	14510	6900	0
1/11/1996	13881	2532	4188	32064	21989	3819	2586
1/18/1996	17181	2410	3519	31917	21851	6182	2190
1/25/1996	13129	1030	4871	41552	17118	3320	1819
2/1/1996	10798	1656	5371	37731	13637	5068	1100
2/8/1996	4377	391	4472	28466	12685	4013	2399

The next step is to upload it to our SQL datawarehouse. This is done by running the following script.

```
Run 12_upload.py
```

The code for upload script is same as in all other cases (with the only exception being the file name and the structure of the tables to be uploaded which needs to be changed each time a new table is uploaded) and therefor left out from this description for brevity.

GrainTransportByMode Table ETL Example

This table represents data for mode of transport (by Truck, Burge or Railroad) of different crops (Corn, Wheat, Soybean, Sorghum, and Barley) in the US, starting from the year 1978. It also segment the data for domestic and export movement.

The following script was used to download the appropriate data file and processed it accordingly.

```
###########################
######Date: 01-18-2016
Copyright: Joshua D. Woodard, Ag-Analytics.org
Contributors: Lin Xue, Tridib Dutta, Josh Woodard, with assistance from Alex Muchocki, Anthony Perello, and Ag-Analytics team.
###########################
######Summary: This script batch read all the fixed width text files into CSV
############################


import pandas as pd
import urllib
```

As before, the following modules were used for our purpose.

Below code snippet retrieves the file from the appropriate URL and saves it in the local folder as `01_output.xlsx'.

```
##retrieve the excel file (put in the code here)
url =
"http://www.ams.usda.gov/sites/default/files/media/DATA%20FOR%20MODAL
%20SHARE%20STUDY%202013.xlsx"
testFile = urllib.URLopener()
testFile.retrieve(url, "01_output.xlsx")
```

Data for different crops were provided in separate tab in the downloaded excel file. So we had to get the excel sheet tab names to automate the process.

```
# these names are retrieved from 01_output.xlsx file above
xls = pd.ExcelFile("01_output.xlsx")
allSheetNames = xls.sheet_names

ExcelSheetName = allSheetNames[1:7]
#ExcelSheetName = ['ALL GRAINS BY MODE ', 'CORN BY MODE','WHEAT
BY MODE','SOYBEANS BY MODE','SORGHUM BY MODE','BARLEY BY
MODE']
```

The following function reads the excel file tab by tab, drop certain unnecessary columns and then combine all the crop information into a single table and saves it as 'GrainTransportByMode.csv' in our local folder for the next step, which is uploading it to our SQL datawarehouse.

For the sake of completeness, we renamed some or all of the columns for the ease of understanding what the columns represent when downloaded by someone who is interested in performing some analysis with these tables.

```
def buildTable(SheetName, inputFileName = "01_output.xlsx"):

   ## if 'ALL GRAINS BY MODE ' is little bit different from the Grain
worksheets themselves
   if SheetName == 'ALL GRAINS BY MODE ':
      DF = pd.read_excel('01_output.xlsx', sheetname = SheetName, skiprows=3)
   else:
      DF = pd.read_excel('01_output.xlsx', sheetname = SheetName, skiprows= 2)

   ## drop the columns 2,4,6
   dropcols = [2,4,6]
   DF.drop(DF.columns[[dropcols]], axis = 1, inplace = True)
```

```
    ## drop the oth row. not needed
    DF.drop(DF.index[[0]], inplace = True)
    DF.index = range(DF.shape[0]) #change the row index
    DF.head()


    Total = DF[:36]
    Export = DF[37: 67]
    Domestic = DF[68:]

    # reset the index (Total's index is already from 0)
    Export.index = range(Export.shape[0])
    Domestic.index = range(Domestic.shape[0])

    # rename the columns
    Total.columns =
['Year','Rail_Total_1000tons','Barge_Total_1000tons','Truck_Total_1000tons']
    Export.columns =
['Year','Rail_Export_1000tons','Barge_Export_1000tons','Truck_Export_1000tons
']
    Domestic.columns =
['Year','Rail_Domestic_1000tons','Barge_Domestic_1000tons','Truck_Domestic_
1000tons']

    ## merge the two columnwise to get one table (colnames indicates Export or
Domestic or total)
    tempDF = pd.concat([Domestic,Export[[1,2,3]]], axis = 1)

    newDF = pd.merge(tempDF, Total, on= 'Year', how = 'outer', left_index =
True)
    newDF = newDF.sort_index()
    fieldName = pd.Series(SheetName.split(" ")[0], index =
range(newDF.shape[0]) )
    newDF = pd.concat([newDF, fieldName], axis = 1)

    return(newDF)

for i in range(len(ExcelSheetName)):
    if i == 0:
        tempDF = buildTable(ExcelSheetName[i])
    else:
        temp = buildTable(ExcelSheetName[i])
        tempDF = pd.concat([tempDF, temp])

tempDF.columns.values[len(tempDF.columns)-1] = 'GrainName'
```

```
tempDF.to_csv("GrainTransportByMode.csv", index = False)
```

Here is a snapshot of the final table.

Year	Rail_Domestic_1000tons	Truck_Domestic_1000tons	Barge_Export_1000tons	Rail_Total_1000tons
1978	NULL	NULL	NULL	117087
1979	NULL	NULL	NULL	127177
1980	NULL	NULL	NULL	143402
1981	NULL	NULL	NULL	127581
1982	NULL	NULL	NULL	121188
1983	NULL	NULL	NULL	130457
1984	66737	86163	60194	124984
1985	64620	103200	51554	105086
1986	80202	102419	45108	115094
1987	93492	117268	56990	139667
1988	94941	121868	58480	151145
1989	92011	89748	62745	143893
1990	92698	111194	62501	134999
1991	85703	128526	63477	126245
1992	94854	115477	68424	135681
1993	91598	136873	60595	134717
1994	96767	124416	57966	124489
1995	101417	139851	67631	152033
1996	84695.756	143425.0132	66920.956	131998.955

Next we upload the document using the script 'upload2DB_GrainTransportByMode.py'. Since this is similar to the upload procedure for the other tables, we leave it out from this report.

GrainTransportCostIndex

Grain Transportation Cost Index table contains data for T & M Grain Transport Cost Index Calculation data. The historical record goes back to 08/21/2002 and contains weekly updates. There are data for Diesel prices, Secondary Unit, Secondary Shuttle, Illinoise River, Ocean Gulf, and Pacific North West (PNW). The cost index is calculated for Trucks (=diesel price, ($/gallon)), for Rail (= near-month secondary rail market bid and monthly tariff rate with fuel surcharge ($/car), for barge (= Illinoise River barge rate), and for Ocean frieght (=Routes to Japan ($/metric ton)). Cost index is calculated taking Year 2000 as the base value.
The script 'GrainTransportCostIndex.py' downloads the data from the USDA website as 'output.xlsx', process it appropriately using pandas data frame and then writes it back to a local folder as '08_processed.csv' file. Then uploads it into the SQL datawarehouse. The procedure is pretty simple as can be seen from below code. The codes are self-explanatory.

```
###########################
Copyright: Joshua D. Woodard, Ag-Analytics.org
```

```
Contributors: Lin Xue, Tridib Dutta, Josh Woodard, with assistance from Alex
Muchocki, Anthony Perello, and Ag-Analytics team.
    ###########################
    ######Summary: This script download the GTRTable1.xlsx from the
    USDA website,
    ######read and process the table using pandas and upload to SQL server
    using bcp(bulk copy).
    import os, sys
    import pyodbc
    import subprocess
    import urllib
    import pandas as pd
```

Above python modules were used for this processing job.

Since we process the data and upload the processed table into a SQL server using a single script, we have the relevant SQL command and the upload command in the script.

```
#global variables
workingDir = "E:\\DatabaseFiles\\UpdateDBFiles\\GrainTransportCostIndex\\"
sqlServerInstance = ".\MSSQLSVRAG" #this way when it is a SQL Server
named instance)
schema="dbo" #schema for processed data
# db="TestDB" #database name
db="AgDB" #database name

#retrieve the file from the link
url = 'http://www.ams.usda.gov/sites/default/files/media/GTRTable1.xlsx'
testFile = urllib.FancyURLopener()
testFile.retrieve(url, workingDir + "output.xlsx")

DF = pd.read_excel(workingDir +'output.xlsx', sheetname = 'Data',
skiprows=range(6),na_values=['nq','n/a','One Week Lag  '] ,parse_dates = True)


##drop the following columns 8,9,15,22,23
dropcols = [7,8,15,22,23]
DF.drop(DF.columns[[dropcols]], axis = 1, inplace = True)

##change the names of the columns
for i in range(7,13):
   DF.columns.values[i] = 'Weekly_Ind' + DF.columns.values[i].split('.')[0]

for i in range(13,19):
   DF.columns.values[i] = 'Base_' + DF.columns.values[i].split('.')[0]
```

```
for i in range(19,21):
    DF.columns.values[i] = 'MonAvgFuelTarrif' + DF.columns.values[i].split('.')[0]

## Extract the date from the Timestamp object
DF.Date = DF.Date.map(pd.Timestamp.date)
DF.to_csv(workingDir + 'processedGTRTable1.csv', index = False)
```

After we process the file, we write the processed file ‘processedGTRTable1.csv’ into our local folder. The next part of our code actually uploads the file in to the SQL server using bulk copy tool (bcp).

```
#### try creating the database if it is already not there
try:
    conn = pyodbc.connect("DRIVER={SQL Server};
SERVER="+sqlServerInstance+"; DATABASE=Master; Trusted connection=
Yes", autocommit = True)
    #Note, default pyodbc connect is autocommit = false. For creating a new
database, must have autocommit = True
##    conn = pyodbc.connect("DRIVER={SQL Server};
SERVER="+sqlServerInstance+"; DATABASE=Master; UID=sa;
PWD=Voshln14!", autocommit = True)
    cursor = conn.cursor()
    query = "CREATE DATABASE " + db
    cursor.execute(query)
    cursor.commit()
    conn.close()
    print "Initial create of ",db," is complete."
except Exception as e:
    print db," DB already exists"
    print e

# ## Creating database is done
# ##########################
conn = pyodbc.connect("DRIVER={SQL
Server};SERVER="+sqlServerInstance+";DATABASE="+db+";Trusted
connection= Yes")
##conn = pyodbc.connect("DRIVER={SQL
Server};SERVER="+sqlServerInstance+";DATABASE="+db+";UID=sa;
PWD=Voshln14!")
cursor = conn.cursor()

try:
 #create table if it does not already exist
```

```
sqlcmd="""CREATE TABLE [dbo].[GrainTransportCostIndex] (
[Date] date,
[Ind_Price] float,
[IndUnit] float,
[IndShuttle] float,
[IndRiver] float,
[IndGulf] varchar(36),
[IndPNW] float,
[Truck] float,
[Unit 1] float,
[Shuttle 1] float,
[Barge] float,
[Gulf 1] float,
[Pacific] float,
[Truck 1] float,
[Unit 2] float,
[Shuttle 2] float,
[Barge 1] bigint,
[Gulf 2] float,
[PNW 1] float,
[MonAvgFuelTarrif_Unit] float,
[MonAvgFuelTarrif_Shuttle] float
)"""
    cursor.execute(sqlcmd)
    cursor.commit()
    conn.close()
    print 'Initial create of [GrainTransportCostIndex] Complete.'
except Exception as e:
    #print '[GrainTransportCostIndex] already exists.'
    sqlcmd="""TRUNCATE TABLE [dbo].[GrainTransportCostIndex]"""
    cursor.execute(sqlcmd)
    cursor.commit()
    conn.close()
    print e

############ use bulk copy (bcp) to load the data into the CommodityFutures table
theproc = subprocess.call('bcp '+db+'.'+schema+'.GrainTransportCostIndex' + ' in ' + workingDir + 'processedGTRTable1.csv' + ' -c -t, -T -S' + sqlServerInstance)
##print theproc
############ Remove the temporary file
os.remove(workingDir+'output.xlsx')
```

Below is a snapshot of the processed data table.

Date	Price	Unit	Shuttle	River	Gulf	PNW
8/21/2002	1.333	-21.5	NULL	128	20.13	11.05
8/28/2002	1.37	-11.5	NULL	128	20.83	11.03
9/4/2002	1.388	-13	NULL	139	22.06	11.21
9/11/2002	1.396	-15	NULL	145	22.7	12.16

RailTraffic table

The source data is over 300 PDF reports we downloaded from website of Association of American Railroads:
https://www.aar.org/newsandevents/Freight-Rail-Traffic/Documents/Forms/CM%20View.aspx

Shown below is a small section of the PDF report.

U.S. Rail Traffic[1]

Week 1, 2016 – Ended January 9, 2016

	This Week		Year-To-Date		
	Cars	vs 2015	Cumulative	Avg/wk[2]	vs 2015
Total Carloads	**239,221**	**-13.5%**	**239,221**	**239,221**	**-13.5%**
Chemicals	32,302	6.2%	32,302	32,302	6.2%
Coal	75,112	-30.7%	75,112	75,112	-30.7%
Farm Products excl. Grain, and Food	16,909	3.5%	16,909	16,909	3.5%
Forest Products	10,656	0.4%	10,656	10,656	0.4%
Grain	21,161	-3.5%	21,161	21,161	-3.5%
Metallic Ores and Metals	19,419	-18.1%	19,419	19,419	-18.1%
Motor Vehicles and Parts	13,276	10.6%	13,276	13,276	10.6%
Nonmetallic Minerals	28,738	-6.5%	28,738	28,738	-6.5%
Petroleum and Petroleum Products	13,096	-15.1%	13,096	13,096	-15.1%
Other	8,552	23.0%	8,552	8,552	23.0%
Total Intermodal Units	**258,939**	**7.5%**	**258,939**	**258,939**	**7.5%**
Total Traffic	**498,160**	**-3.7%**	**498,160**	**498,160**	**-3.7%**

[1] Excludes U.S. operations of CN and Canadian Pacific.
[2] Average per week figures may not sum to totals as a result of independent rounding.

The PDF convertor we use is Xpdf: http://www.foolabs.com/xpdf/home.html
In particular, we used the pdftotext.exe binary file for Windows system, which is part of the Xpdf.
After Xpdf is installed or binary files of pdftotext.exe is downloaded, the command to convert PDF to text files is:

```
pdftotext –table file.pdf
```

It will create a file.txt file in the same directory.
The ETL process:

Step 0: Run InitLoad_PDFstoCSV.py to batch convert all the PDF files to txt files using Xpdf’s pdftotext.exe then read the text files into one csv file. Run Upload_RailTraffic.py to upload the data table dbo.RailTraffic to SQL server.
This step only needs to be done once to create the 04_railtraffic.csv for initial load. All future updates will append newer records to existing dataset.
Step 1: Run Update_railtraffic.py to download the newest PDF file from source URL, convert it to text file using using Xpdf’s pdftotext.exe, read it and append the data to 04_railtraffic.csv.
Step 2: Run Upload_RailTraffic.py to update the data table dbo.RailTraffic
Shown below is the python script to update RailTraffic dataset.

```
###########################
######Date: 01-21-2016
Copyright: Joshua D. Woodard, Ag-Analytics.org
Contributors: Lin Xue, Tridib Dutta, Josh Woodard, with assistance from Alex Muchocki, Anthony Perello, and Ag-Analytics team.
###########################
######Summary: This script download the newest PDF from https URL, convert it to text file, read it and append the data to railtraffic.csv
############################
import os, sys
import os.path
import urllib2
import ssl, socket
import itertools
import re

#global variables
workingDir = "E:\\DatabaseFiles\\UpdateDBFiles\\RailTraffic"
mypath = 'https://www.aar.org/newsandevents/Freight-Rail-Traffic/Documents/Forms/CM%20View.aspx'
#note: it's https
# ssl.create_default_context module only available after python2.7.9
ctx = ssl.create_default_context()

# define function to fetch the newest PDF link from a webpage
def fetchPDFlink(url):
   try:
      mylines = urllib2.urlopen(url, context=ctx).readlines() # note: context parameter has to be passed for https
      k = re.search('href="(\S+).pdf"', ''.join(mylines))
      pdflink = k.group(0)
      pdflink = pdflink.replace('href="', 'https://www.aar.org')
      pdflink = pdflink.replace('"', '')
      return pdflink
   except Exception as e:
      print "NOT found plz check url"
      print e
```

```
# define function to download the PDF from the link and convert it to text file
def downloadPDF(pdflink, filename):
   try:
      webFile = urllib2.urlopen(pdflink, context = ctx)
      pdfFile = open(workingDir + "\\" + filename, 'wb')
      pdfFile.write(webFile.read())
      webFile.close()
      pdfFile.close()
      # base = os.path.splitext(fname)[0]
      # os.rename(fname, base+ ".pdf")
   except IOError as e:
       print e
   #convert PDF to txt
   os.system( workingDir + "\\pdftotext -table " + workingDir + "\\" + filename)


# function to check if a string can be represented as a number
def is_number(s):
   try:
      float(s)
      return True
   except ValueError:
      return False

# function to read a text file and append a new line in output csv
def read_txt(filename):
   # grab the date from the text file name
   date = filename[:10]
   # open the input file with read only permit
   with open (workingDir + "\\"+ filename) as inF:
      outF.write(date + ",")
      linestr = ""
      for line in itertools.islice(inF, 8, 33):
         #strip the newline character
         line = line.rstrip()
         #check to see if the line is empty
         if line:
            #remove all comma in the line
            line = line.replace(",", "")
            arr = re.split("\s+", line)
            for ii in arr:
               # call function to check if ii is a number
               if is_number(ii):
                  linestr += ii + ","
                  break
```

```
    linestr = linestr.rstrip(",")
    outF.write(linestr + "\n")
###############
# main()
###############
# open an output file to append
outF = open(workingDir + "\\04_railtraffic.csv", "a")
# call function to fetch the newest PDF link from a webpage
newlink = fetchPDFlink(mypath)
#get filename from the pdflink
fname = newlink.split('/')[-1]
#textfile name
textfile = fname.replace(".pdf", ".txt")
#check to see if file exists in current workingDir
if not os.path.isfile(fname):
  # call function to download the PDF and convert it to text file
  downloadPDF(newlink, fname)
  # call function read_txt to read the text file and append new line to
04_railtraffic.csv
  read_txt(textfile)
outF.close()
```

Shown below is a small section of the final data table. The rows of carloads types in the raw PDF file are transposed to be the columns of the final table. Each row in the final table represent data from one raw PDF file. Data from the PDF files are appended one after another to form a time series.

Date	Total Carloads	Chemicals	Coal	Farm and Food Products	Forest Products	Grain	Metallic Ores and Metals	Motor Vehicles and Parts	Nonmetallic Minerals and Products
3/7/2013	283819	31360	114155	16456	10963	17289	25177	17803	28501
3/14/2013	276698	29196	112000	16862	10769	17625	21106	18182	29936
3/21/2013	280624	30143	111302	16110	10585	17379	23030	19430	31343
3/28/2014	278738	30460	110013	16415	11171	17034	23517	17561	32279
4/4/2013	281367	30557	109700	15899	11742	15388	28041	16906	31684
4/11/2013	280748	30475	111153	16365	11039	16888	22918	16502	33911
4/18/2013	275675	29609	104028	16449	11198	17150	23323	17913	34959
4/25/2013	276662	29598	106728	16331	10600	15670	25071	17430	34261
5/2/2013	275638	29891	104807	16000	10791	15672	25599	17294	34163

Set up SQL Server Agent Job to update this dataset weekly.
Step 1: C:\python27\python E:\DatabaseFiles\UpdateDBFiles\RailTraffic\Update_railtraffic.py
Step 2: C:\python27\python E:\DatabaseFiles\UpdateDBFiles\RailTraffic\Upload_RailTraffic.py
Schedule: Weekly on Friday at 1 am

Every time the update happens, the newest PDF report is downloaded and its data is parsed and extracted to form one new record to be appended to the data table.

SecondaryRailcarBids

Secondary Rail Car Bids

Data for the rail car bids in the secondary market is available for two US railroad companies: Burlington Northern Sante Fe (BNSF) and Union Pacific (UP). The weekly data is available from 5/3/1997 until now.

The script 'SecondaryRailCarbids.py' downloads the raw data file into a local folder, process it, writes the processed file into local folder as '07_processed.csv' and then uploads it to a SQL datawarehouse using bulk copy tools (bcp).

Below is a snapshot of the code for the processing job.

```
##########################
###### Date: 01-22-2016
Copyright: Joshua D. Woodard, Ag-Analytics.org
Contributors: Lin Xue, Tridib Dutta, Josh Woodard, with assistance from Alex
Muchocki, Anthony Perello, and Ag-Analytics team.
##########################
###### Summary: This script download the GTRFigure4-6.xlsx from the ams
website:
###### http://www.ams.usda.gov/services/transportation-analysis/gtr-datasets,
###### read and process the table using pandas and upload to SQL server using
bcp(bulk copy).
###########################
import os, sys
import pyodbc
import subprocess
import urllib
import pandas as pd

#global variables
workingDir = "E:\\DatabaseFiles\\UpdateDBFiles\\SecondaryRailcarBids\\"
#workingDir =
"C:\Users\lx58\Dropbox\AgDB_Admin\OCE\data_summaries\Railway_Database
"
sqlServerInstance = ".\MSSQLSVRAG" #this way when it is a SQL Server
named instance)
#sqlServerInstance = "AG-AEM-1M9RBY1" #this way when it is not a SQL
Server named instance)
#sqlServerInstance = "AG-AEM-6656V12\MSSQLAGDEV1" ## mine is sql
named server instance
```

```
schema="dbo" #schema for processed data
#db="TestDB" #database name
db="AgDB" #database name

#retrieve the file from the link
url = 'http://www.ams.usda.gov/sites/default/files/media/GTRFigure4-6.xlsx'
testFile = urllib.FancyURLopener()
testFile.retrieve(url, workingDir + "07_output.xlsx")

DF = pd.read_excel(workingDir + '07_output.xlsx', sheetname = 'Secondary')
DF.to_csv(workingDir + '07_processed.csv', index = False)
```

The below script updates the processed file into a SQL server.

```
########################
#### try creating the database if it is already not there
try:
   conn = pyodbc.connect("DRIVER={SQL Server};
SERVER="+sqlServerInstance+"; DATABASE=Master; Trusted connection=
Yes", autocommit = True)
   #Note, default pyodbc connect is autocommit = false. For creating a new
database, must have autocommit = True
   cursor = conn.cursor()
   query = "CREATE DATABASE " + db
   cursor.execute(query)
   cursor.commit()
   conn.close()
   print "Initial create of ",db," is complete."
except Exception as e:
   print db," DB already exists"
   print e

# ## Creating database is done
# ##########################
conn = pyodbc.connect("DRIVER={SQL
Server};SERVER="+sqlServerInstance+";DATABASE="+db+";Trusted
connection= Yes")
cursor = conn.cursor()

try:
 #create table if it does not already exist
   sqlcmd="""CREATE TABLE [dbo].[SecondaryRailcarBids] (
[Week Ending] date,
[Bid Month] varchar(10),
[Month Number] bigint,
[Bid Year] bigint,
[Company] varchar(8),
```

```
[For Search] varchar(16),
[Non-Shuttle] float,
[Shuttle] float
)"""
    cursor.execute(sqlcmd)
    cursor.commit()
    conn.close()
    print 'Initial create of [SecondaryRailcarBids] Complete.'
except Exception as e:
    #print '[GrainTransportCostIndex] already exists.'
    sqlcmd="""TRUNCATE TABLE [dbo].[SecondaryRailcarBids]"""
    cursor.execute(sqlcmd)
    cursor.commit()
    conn.close()
    print e

############ use bulk copy (bcp) to load the data into the CommodityFutures table
theproc = subprocess.call('bcp '+db+'.'+schema+'.SecondaryRailcarBids' + ' in ' + workingDir + '07_processed.csv' + ' -c -t, -T -S' + sqlServerInstance)
##print theproc
############ Remove the temporary file
os.remove(workingDir+'07_output.xlsx')
```

A snapshot of the processed data table is provided below.

Week Ending	Bid Month	Month Number	Bid Year	Company	For Search	Non-Shuttle	Shuttle
5/3/1997	January	1	1998	BNSF-GF	355531BNSF-GF	NULL	NULL
5/3/1997	January	1	1998	UP-Pool	355531UP-Pool	NULL	NULL
5/3/1997	February	2	1998	BNSF-GF	355532BNSF-GF	NULL	NULL
5/3/1997	February	2	1998	UP-Pool	355532UP-Pool	NULL	NULL
5/3/1997	March	3	1998	BNSF-GF	355533BNSF-GF	NULL	NULL
5/3/1997	March	3	1998	UP-Pool	355533UP-Pool	NULL	NULL
5/3/1997	April	4	1998	BNSF-GF	355534BNSF-GF	NULL	NULL
5/3/1997	April	4	1998	UP-Pool	355534UP-Pool	NULL	NULL
5/3/1997	May	5	1997	BNSF-GF	355535BNSF-GF	-95	NULL
5/3/1997	May	5	1997	UP-Pool	355535UP-	-28	NULL

					Pool		

TrainSpeedByCompany

Train speed data is available from American Association of Railroads (AAR) website. The publicly available data is for 2015 only. The historical data is hidden behind paywall.
The weekly data is reported by the six major U.S. railroad companies and contains train speed data among other performance measures. We collected the train speed data for our data warehouse.
Note that the data couldn't be programmatically downloaded from AAR's website. Instead, we downloaded it manually and then processed it using the following python script.

```
## the url of the file is hidden and cannot be retrieved. So I guess we have to download the file manually
## The manually downloaded file is saved as 05_TrainSpeed.csv in the Railway Database under AgDB folder in dropbox
import csv
```

The above python module is used to process the data.
Below is a utility function which we will use later.

```
## define is_number() function
def is_number(s):
    try:
        float(s)
        return True
    except ValueError:
        return False
```

Below is the code snippet to process the data.

```
#get the dates
with open('05_TrainSpeed.csv', 'rb') as f:
    reader = csv.reader( (line.replace('\0','') for line in f) )
    count = 0
    for row in reader:
        my_string = ' '.join(row)
        if count ==1:
            break
        elif 'Railroad Measure Category' in my_string:
            a = my_string
            a = a.strip().split()
            #print a
```

```
            tr = []
            date = []
            for item in a:
                if item.isalpha():
                    tr.append(item)
                elif is_number(re.sub('[^a-zA-Z0-9 ]','',item)):
                    date.append(item)
            tr = [' '.join(tr)]
            #print tr + ['4Q14']+ date
            count = count + 1

f.close()


A = []
#add the date information to the first line of A
A.append(tr + ['4Q14']+ ['Dec'+ date[0]] + date[1:])

# retrieve the rest of the data
companyTrainCarType = ['BNSF','CN','CSX','Kansas','Norfolk','Union']

with open('05_TrainSpeed.csv', 'rb') as mycsv:
        reader = csv.reader( (line.replace('\0','') for line in mycsv) )
        for row in reader:
            new_string = ' '.join(row)

            new_string = re.sub('[^a-zA-Z0-9\n\. ]','', new_string) ## clean up the bad
characters

            if not re.sub('[^a-zA-Z0-9]','',new_string).isalpha():
                if re.search("Train Speed MPH", new_string):
                    for name in companyTrainCarType:
                        if name == 'Kansas':
                            new_string = re.sub('S.A. de C.V.','',new_string)
                        if re.search(name, new_string):
                            #new_list = re.sub('[^0-9\n\. ]','',new_string).strip().split()
                            b = []
                            num = []
                            for term in new_string.strip().split():
                                #print term
                                if is_number(term):
                                    num.append(term)
                                elif term.isalpha():
                                    b.append(term)
                            b = ' '.join(b)
                            b = re.sub('Train Speed MPH','', ''.join(b))
```

```
            b = [b]
            A.append(b + num)


mycsv.close()

##now write the transpose of this A to a .csv file
import csv
B = zip(*A) #transpose the matrix
with open("05_processed.csv", "wb") as correct:
    writer = csv.writer(correct)
    writer.writerows(B)
correct.close()
```

Below is a snapshot of the processed data set.

Railroad Measure Category	BNSF Intermodal	BNSF Manifest	BNSF Coal Unit
1/9/2015	35.3	22.3	17.6
1/16/2015	34.6	21.8	18.8
1/23/2015	34.5	21.7	19.4
1/30/2015	33.3	21.6	19.3
2/6/2015	32.8	20.6	17.6
2/13/2015	34.2	21.1	18.4

The upload proceedure is similar to the other datasets and left out from this report for brevity.

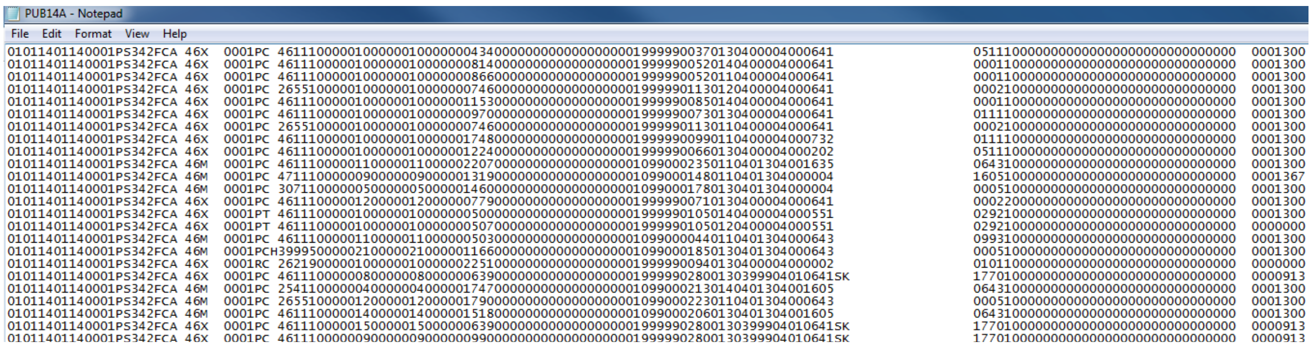
2.12 WaybillSamples

The Public Use Waybill Sample (PUWS) is a non-proprietary version of the STB Carload Waybill Sample. The STB collects the data under the requirements that all US railroads that terminate more than 4,500 revenue carloads must submit a yearly sample of terminated waybills. Samples are available annually from 2000 to 2014. The source is: http://www.stb.dot.gov/STB/industry/econ_waybill.html

Revised 2014 Public Use Waybill Sample
2013 Public Use Waybill Sample
2012 Public Use Waybill Sample
2011 Public Use Waybill Sample
2010 Public Use Waybill Sample
2009 Public Use Waybill Sample
2008 Public Use Waybill Sample
2007 Public Use Waybill Sample
2006 Public Use Waybill Sample
2005 Public Use Waybill Sample
2004 Public Use Waybill Sample

[2003 Public Use Waybill Sample](#)
[2002 Public Use Waybill Sample](#)
[2001 Public Use Waybill Sample](#)
[2000 Public Use Waybill Sample](#)

Each Waybill Sample is a text file that is unable to read without parsing the data first. See below:



PUB14A - Notepad

File Edit Format View Help

```
01011401140001PS342FCA 46X  0001PC 4611100000100000010000000434000000000000000000199999003701304000040064 1    05111000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 46111000001000000100000008140000000000000000001999990052014040000400641    00011000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 46111000001000000100000008660000000000000000001999990052011040000400641    00011000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 26551000001000000100000007460000000000000000001999990113012040000400641    00021000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 46111000001000000100000011530000000000000000001999990085014040000400641    00011000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 46111000001000000100000009700000000000000000001999990073013040000400641    01111000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 26551000001000000100000007460000000000000000001999990113011040000400641    00021000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 46111000001000000100000017480000000000000000001999990099011040000400732    01111000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 46111000001000000100000012240000000000000000001999990066013040000400202    05111000000000000000000000000000000  0001300
01011401140001PS342FCA 46M  0001PC 4611100000110000011000002207000000000000000000109900235011040130400163 5    06431000000000000000000000000000000  0001300
01011401140001PS342FCA 46M  0001PC 4711100000090000009000001319000000000000000000109900148011040130400000 4    16051000000000000000000000000000000  0001367
01011401140001PS342FCA 46M  0001PC 3071100000050000005000001460000000000000000000109900178013040130400000 4    00051000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 4611100000120000012000000779000000000000000000199999007101304000040064 1    00022000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PT 4611100000100000010000000500000000000000000000199999010501404000040055 1    02921000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PT 4611100000100000010000000507000000000000000000199999010501204000040055 1    02921000000000000000000000000000000  0000000
01011401140001PS342FCA 46M  0001PC 4611100000110000011000000503000000000000000000109900044011040130400064 3    09931000000000000000000000000000000  0001300
01011401140001PS342FCA 46M  0001PCH3999500000210000021000001166000000000000000000109900185013040130400064 3    00051000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001RC 2621900000100000010000002251000000000000000000199999009401304000040000 2    01011000000000000000000000000000000  0000000
01011401140001PS342FCA 46X  0001PC 461110000008000000800000063900000000000000000019999902800130399904010641SK    17701000000000000000000000000000000  0000913
01011401140001PS342FCA 46M  0001PC 2541100000040000004000001747000000000000000000109900213014040130400160 5    06431000000000000000000000000000000  0001300
01011401140001PS342FCA 46M  0001PC 2655100000120000012000001790000000000000000000109900223011040130400064 3    00051000000000000000000000000000000  0001300
01011401140001PS342FCA 46M  0001PC 4611100000140000014000001518000000000000000000109900206013040130400160 5    06431000000000000000000000000000000  0001300
01011401140001PS342FCA 46X  0001PC 461110000015000001500000063900000000000000000019999902800130399904010641SK    17701000000000000000000000000000000  0000913
01011401140001PS342FCA 46X  0001PC 461110000009000000900000099000000000000000000019999902800130399904010641SK    17701000000000000000000000000000000  0000913
```

The Surface Transportation Board provides a Reference Guide to help understand the raw data.
http://www.stb.dot.gov/STB/docs/Waybill/2014%20STB%20Waybill%20Reference%20Guide.pdf

Page 99-100 of this Reference Guide contains Table 4-6. 247-Byte STB Public Use Waybill File Record Layout. Basically, each row is a record. Each record contains fixed-width data. 1–6 byte is Waybill Date, 7-10 byte is Accounting Period, and 11-14 byte is Number of Carloads, so on and so forth till 247 byte.

By reading the PDF Reference Guide comes with the sample and writing Python scripts to parse the data and assign them to separate fields, we are able to decipher the raw code into readable data in CSV format. (We can get data such as commodity code, carloads, Tons)

	A	B	C	D	M	N	O	P	AF	AG	AQ	AR	BH	BI	BJ	BK
1	Waybill Date	Accounting Period	Num Of Carloads	Car Ownership Code	Commodity Code(STCC)	Billed Weight In Tons	Actual Weight In Tons	Feight Revenue	Origin BEA Area	Origin Freight Rate Territory	Termination BEA Area	Termination Feight Rate Territory	Expanded Carloads	Expanded Tons	Expanded Feight Revenue	Expanded Trailer/Container Count
2	10114	114	1	P	46111	10	10	434	64	1	51	1	40	400	17360	40
3	10114	114	1	P	46111	10	10	814	64	1	0	1	40	400	32560	40
4	10114	114	1	P	46111	10	10	866	64	1	0	1	40	400	34640	40
5	10114	114	1	P	26551	10	10	746	64	1	0	2	40	400	29840	40
6	10114	114	1	P	46111	10	10	1153	64	1	0	1	40	400	46120	40
7	10114	114	1	P	46111	10	10	970	64	1	11	1	40	400	38800	40
8	10114	114	1	P	26551	10	10	746	64	1	0	2	40	400	29840	40

Below is the python script snippet we wrote to parse the raw data. We used python module pandas to read the fixed width text file into CSV format.

```
import sys, os
import numpy
import pandas as pd

fwidths =
[6,4,4,1,4,4,2,3,4,1,1,1,5,7,7,9,9,9,1,1,1,1,1,1,1,4,1,1,5,3,1,3,1,2,2,2,2,2,2,2,2,2,2,3,
1,1,5,3,4,5,4,4,4,1,1,2,1,4,46,1,6,9,11,6]
dateparse = lambda x: pd.datetime.strptime(x, '%m%d%y')
# define function to read file using pandas
def readfiles(filename):
```

```
    df = pd.read_fwf(filename, widths = fwidths, names = colnames,
parse_dates=['Waybill Date'], date_parser =dateparse, converters={'Accounting
Period': str})
    #define name of the processed csv
    csvname = filename.split(".")[0]+ ".csv"
    #write to csv file
    df.to_csv(csvname, index = False)
```

The ETL process:
Step 0: Use ReadWaybill.py to parse the raw text files into readable CSV and add header line for the fields.
Step 1: Clean up raw data using cleanFields.py
Step 2: Combine all cleaned csv files into 02_processed.csv using buildTable.py
Step 3: upload data into SQL server using upload2DB.py
(note: If Step 1 indicate columns that have different data type than what SQL import wizard suggested, manually change those to varchar when creating the CREATE TABLE statement.)

Step 1 Details:
The Waybill samples contain many columns of mixed type data (string mixed with numbers). Each year, the mixed type columns vary. So each CSV file has to be examined separately before creating the final "02_processed.csv".
Three kinds of actions are performed:

- If the column contains both string and number, we assign it varchar type when uploading to SQL server, thus nothing needs to be done during preprocessing.
- If the column contains a mistake (e.g. a '10' (str) among 10 (int)), we wrote script to force convert the string type to number.
- If the column contains meaningless special symbols (such as '*****'), we wrote script to replace them with blank cells (NULL).

```
# PU2005
[Col-14: Actual Weight In Tons]: replace "*******" with NULL and force
convert to int type.
[Col-36,37,38, 53: Interchange State #4, #5, #6, Num of Axles]: varchar

# PUB06A
[14: Actual Weight In Tons]: replace "*******" with NULL and force convert to
int type.
[33, 34, 35, 36, 37, 53: Interchange State #1, #2, #3, #4, #5, #6, Num of Axles] :
varchar

# PUB07A
[35,36,53]: varchar

# PUB08A
[35,36,37,53]: varchar
```

```
#PUB09A
[28: Exact Expansion Factor]: replace "*****" with NULL and force convert to int type.
        [Row-60443:Col-28]: "*****"
        [Row-60447:Col-28]: "*****"
        [Row-60448:Col-28]: "*****"
[33,34,35,36,37,53]: varchar

# PUB10A
[11:Hazardous/Bulk Material In Boxcar]: varchar
[35,36,53]: varchar

# PUB11A
[35, 36, 37, 38]: varchar

# PUB12A
[7: TOFC/COFC Service Code]: varchar
[9: Trailer/Container Ownership Code]: varchar
[10: Trailer/Container Type Code]: varchar
[13: Builled Weight in Tons]:  int
[11, 33, 34, 35, 36, 37, 53]: varchar

# PUB13A
[35,36,37,53]: varchar

# PUB14A
[35,36,37]: varchar
```

DIFFICULTIES AND RECOMMENDATIONS

OBSERVATIONS DURING THE ETL PROCESS

Several observations were made during the ETL process of processing these datasets for use. First, among the various raw data formats, CSV and XLSX (Excel) are in general the easiest to parse, followed by XML and fixed width text. While PDF is a nice format for displaying, it is well accepted that it is not recommend for data storage. More often than not, the table layout is lost during the conversion from PDF to text file and the resulting data cannot be parsed easily. Data that is not in a format of delimited or fixed width cannot be parsed by programming language easily, hence make data automation difficulty unnecessarily by the government and other data publishing entities. Second, for web scrapping purpose, it is much easier to scrape files that follow common naming convention and if the URL is persistent both in form and naming convention. Despite the obviousness of this statement, it is only occasionally followed, and disingenuous responses from data publication agencies (or no response) is not uncommon when this is suggested, requested, or pointed out.

A good example though is the *ExportGrainTotals* dataset. A new annual CSV data file named CYyyyy.csv (ie. CY2016.csv) is posted under the same URL consistently: https://www.gipsa.usda.gov/fgis/exportgrain/

- Export Grain Inspection 2016 (last updated 3/7/2016 4:20:58 PM)
- Export Grain Inspection 2015 (last updated 2/16/2016 7:01:47 AM)
- Export Grain Inspection 2014 (last updated 5/26/2015 4:34:21 PM)
- Export Grain Inspection 2013 (last updated 6/20/2014 8:22:09 AM)
- Export Grain Inspection 2012 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2011 (last updated 4/8/2013 12:00:00 AM)
- Export Grain Inspection 2010 (last updated 4/8/2013 12:00:00 AM)

Third, when data entry is done by different railroad companies, they come up with their own data forms and making it difficult to combine and summarize the data later. It would contribute to taxpayer value if the government agent were to provide the companies with a template. Fourth, we noticed many human errors during the process converting the raw data. The most common issue is that columns of Excel files have mixed data type. For example, it is not uncommon to find a column of float data type, but then one cell among a million or so that randomly contains special characters such as "*****". It is then necessarily to programmatically detect these outliers and manually clean up those cells before uploading. It is recommended that some data validation control being implemented during data entry to minimize human errors at source.

FreightCommodityStatistics: A Short Case

During the ETL process of the *FreightCommodityStatistics*, we encountered the following issues:

1. The url for some of the excel raw data file seems to have changed very recently and URLopener() function from the `urllib` module no longer works. We needed to replace `URLopener(`) by `FancyURLopener()` to handle the redirect. There was no announcement or tracking of this issue. Thus, centralization and standardization is likely beneficial here.
2. Several of the excel files for companies including BNSF, GTC, etc. have multiple format errors rendering them overly burdensome to convert to flat files (`.csv`). All the scripts above in Step 1, contain a function called `clean_cell()` which addresses many of those formatting abnormalities buried deep inside those excel raw data files, but not all. For example, in the raw data file for the year 2012 from the company BNSF, most of the entered numbers had space in them (`2124 4` instead of 21244) making them read in as `string` in the database or python reader, resulting in obvious misinterpretation while performing analysis or any mathematical operations on these datasets. Some of the cells seems to have accidentally encoded as a `datetime` object when they should be clearly numeric. Our `clean_cell()` is able to handle all most of these problems, but again had to be implemented and developed by us internally, and the next team will surely unnecessarily run up to the same issues. This is wasteful, and the solution is likely some basic standardization. And note still that there could likely be unexpected formatting errors in the file which `clean_cell()` in the future if formats again change unnecessarily from source.
3. Another major issue we ran into is the encoding of different goods/Commodities that the railway companies transported. It seems that there is a industry-wide standard for encoding commodity goods the railway companies handle. They have assigned integer code to different goods. For example the number 1 (sometimes entered as 01, although not consistently) represents a broad category which is called 'Farm Products'. Under this category is 'Field Crops',

which is assigned a value 11 (sometimes entered as 011, , although not consistently). Ironically, in the same document, the number 11 was assigned to 'Coal' which is a broad category representing coal based products such as Anthracite which was assigned a numeric value of 111. This we thought could create a lot of confusion especially if this coding is used as 'key' while downloading data from our database. To resolve the matter, we had to come up with our own recoding of these numeric values which necessitated explicitly encoding a sequential record recognition routine. We decided that instead of using integer values, we would use float type numbers and 1 will be encoded as 1.000 while 11 (representing Field Crops, a sub category of Farm product) will be assigned the value 1.100 instead of 11 or 011. This way coal can be 11.000 and so on. We wrote a function called `recode()` which can be found in each of the scripts mentioned in Step 1.