# Introduction

Xero’s Small Business Insights (XSBI) is an ititiative to deliver aggregated business and economic statistics from Xero customer data that reflect the performance of the small business economy. To fulfill their purpose and be valuable for and trusted by their audience, these statistics and any analysis built on top of them need to be produced in a systematic and robust way. The XSBI Pipeline is three interconnected things:

1. **the framework** that conceptually organises the production process of XSBI data products,
2. **the code base** that implements the framework and embodies the actual XSBI production process,
3. **the data products**, both final and intermediate, that are delivered by the XSBI production process.

This document provides an overview of all three aspects of the XSBI Pipeline describing the rationale for and the implications of choices made in designing the production process, detailing how the code is structured and run, and what key outputs are.

# The framework

In essence, XSBI is a production process that takes Xero customer data as input and transforms it into a range of final data products as output. Input is typically highly granular transactional data *extracted* from the Xero product (e.g. data relating to individual invoices, payslips, movement in individual accounts). Output is typically a highly aggregated statistic.

XSBI originally started out with a suite of four business statistics. Each statistic was produced by way of a stand-alone SQL script that did all necessary processing in one go. While that may have been computationally efficient, it certainly made it much more difficult to assess and ascertain that the output matched exactly what was intended.

The XSBI Pipeline adopts a different approach. It identifies a general pattern of tasks or transformations that are required to produce any final aggregate statistic and rather than entangling all necessary steps into a monolythic ‘black box’ script, the steps are executed in a neat isolated way sequantially leading to important intermediate data products along the way, which are stored and used for multiple downstream purposes.

## The general pattern

XSBI fundamentally deals with two different units of analysis representing the *micro* level: (1) business transactions related to (2) organisations. *Organisations* are Xero customers who store and maintain their accounting records on the Xero platform. A single *organisation* is currently assumed to correspond to a single *small business* in the real economy.

With a few exceptions, XSBI receives unaggregated transactional data (*source data*) extracted from the Xero product. In general, the product does not store aggregations internally. Individual users trigger such functions at run-time for their own purpose (e.g. an account balance or a financial report is not pre-calculated and stored, but calculated/compiled from all available transactional data from the ground up).

Accordingly, the XSBI source data needs to be processed into a range of datasets corresponding to accounting or business concepts meaningful at either the transaction or organisation level. These intermediate datasets, often sharing common ‘parent’ datasets, are then used to produce the high-level aggregate statistics, the final output of the XSBI Pipeline.

This process can be divided into the following conceptual steps:

|  |  |  |  |
| --- | --- | --- | --- |
| Step | Name | Description | Example |
| 1 | Primary filtering | Subset records using criteria applicable to the transaction level | Select all AR invoices |
| 2 | Primary aggregation | Aggregate records at the organisation level | Calculate monthly credit sales for each organisation |
| 3 | Secondary filtering | Subset records using criteria applicable to the organisation level | Select (sample) monthly credit sales records for orgs that meet SBI criteria |
| 4 | Secondary aggregation | Aggregate records into high-level statistic | Calculate average monthly level of credit sale for the sample |

### Primary filtering

Filtering transactional records can serve two purposes: (1) conceptual validation and (2) data validation. From a data processing point of view, filtering reduces datasets row-wise by way of a WHERE SQL query clause (or the equivalent).

#### Conceptual validation

The wealth of transactional data makes it possible to measure a wide range of interconnected quantitative accounting concepts at scale. Depending on the content and definition of these concepts, transactional data needs to be aggregated selectively. The main purpose of primary selection is to subset records and only retain those in a dataset that are in line with a particular concept or set of concepts.

As accounting concepts tend to build upon each other, it often makes sense to perform primary filtering in separate consecutive steps, so that intermediate datasets can serve multiple avenues of processing and thus task duplication is avoided.

**Example**

The transactional records ultimately used to calculate *Getting paid* statistics (e.g. Average Days to Pay) are selected from the invoices dataset in two filterings steps. In the first step, we select all accounts receivable invoices and retain them as a dataset (AR\_invoices). In the second step, we further subset the accounts receivable invoices dataset by selecting all invoices that have been already paid (paid\_AR\_invoices). As a benefit, the in-between AR\_invoices dataset can also be used for calculating monthly credit sales at the organisation level.

#### Value validation

XSBI source data may contain a range of different errors (See the Data Quality Framework for details). Some forms of errors, such as invalid cell values can be easily detected and be dealt with. The simplest approach is to filter out records containing invalid values early on in the primary filtering step.

**Example**

The statistic *Average Overdue Days* calculates how many days late invoices are paid on the average. It is based on the difference between the date of full payment and the due date, two attributes potentially available for each paid invoice on record. However, the Xero software does not validate dates and so the user can record due dates and received payment dates freely. While most users may be accurately entering those dates, our dataset contains invoices with invalid values in these critical variables.

Dates can be simply out of range (e.g. 1st January 1867 or 3rd August 2064) or be internally inconsistent: the due date preceding the invoice date or the date of full payment preceding the invoice date. During primary filtering records containing such invalid values may be chosen to be filtered out.

### Primary aggregation

The purpose

Self-assessment (Click to expand)

* Abc
* Abc

## Stepwise processing

The pipeline processes source data into final data products (aggregate metrics) in several consecutive steps rather than in one go. The output from each step is retained as an intermediate dataset which is then used as input in the next step. This provides the following benefits: 1. **Interpretability**. Data processing is not a ‘black box’: it is broken out into a series of logical steps corresponding more directly to the intuitive analytical/research workflow. It makes it easier to follow what exactly happens between source data going in and final data products coming out of the pipeline. 2. **Transparency**. Given that intermediate datasets are retained, quality control and assurance measures are much easier to put in place at each stage of processing to monitor errors and anomalies and to provide data lineage information. 3. **Consistency**. With production following the same pattern for all output, it is easier to maintain consistency not only for products individually but across a whole suite of them, 4. **Expandability**. With intermediate products exposed, it is much easier to ‘tap’ the pipeline rather than having to start data processing from scratch for a new metric or piece of analysis. With documented content and known provenance, intermediate data products will provide a convenient ‘entry point’ for new/additional processing pipes.

## Separated org-level filtering

Sampling can happen on two levels: (1) transaction/document level and (2) organisation level. Sampling is effectively a filtering task where records are subset so that only those that meet requirements (~should be *in* the sample) are retained for further analysis. (Currently, sampling at both levels is done using a full enumeration rather than a probabilistic approach.) In the name of the ‘separation of concerns’, the XSBI Pipeline takes care of transaction/document-level filtering separately from org-level filtering. This means that intermediate data products would only involve filters on transactions/documents and potentially primary aggregates (e.g. a balance), and org-level subsetting would always be the last step before final aggregation. In other words, any intermediate datasets before that would contain data for the complete org set.

This approach provides flexibility around working with alternative sampling approaches and/or sample inclusion rules. SBI, for instance, currently considers orgs with at least 12 months of tenure on the Xero platform. If org-level filtering happened in the very beginning (maybe even intermingled with transaction-level filtering), then even a minor experimental tweak such as relaxing the requirement to only 6 months on Xero would require running the pipeline from the very beginning. With *last minute* org selection this will not be a problem.

# Running environment

* **Presto**. Source data tables are made available by Data Services in Caspian/Hive/Presto. This pipeline reads the source data tables and writes intermediate tables (~datasets) back into Presto, using a the sbi\_temp schema. The pipeline currently ends with aggregate metrics kept in memory (R), but the pipeline is soon to be extended to ‘publish’ these statistics to the XSBI statistical database.
* **R/SQL**. As relying on Presto, the code is composed of a bunch of parameterised SQL queries. The parameters are injected from R, where the ‘pipelining’ script runs.
* **SBI package**. The R scripts rely on a package called sbi, which provides the functions needed to run the pipeline, such as connecting to Presto, handling the key parameters, loading and executing the snippets, producing a ‘pipeline flow diagram’. This package needs to be installed first (repo coming soon).

# Code structure

The pipeline consists of processings steps that turn source (transactional) data into various aggregate business and economic statistics. The process involves joining fragmented transactional data into a unified ‘view’ in which each row corresponds to one transaction (~‘case’). Subsequent processing steps involve subsetting (filtering) records to meet business definitions, calculating new indicators at the record level (mutation), and performing varous levels of aggregation (summarising). The pipeline deliberately breaks data processing operations into a chain of distinct steps. Each step performs one well-defined function. This is done in an effort to keep the joining/filtering/mutation/aggregation separate for the sake of transparency, more closely following the actual logical steps of working with a dataset rather then driven by query optimisation which may lead to effcient SQL scripts which are difficult to untangle for debugging and modification.

The code consists of a main runnable R script (xsbi-core-pipeline), that chains together the processings steps in a series of calls to subordinated R scripts (sitting in the folder steps/) that currently all wrap Presto SQL queries (sitting in steps/sql/), but future processing steps may be more complicated and reliant on other tools & computational resources.

xsb-core-pipeline.R  
▗▅▅▅▅▅▅▅▅▅▅▅▅▅▅▅▅▅▅▖   
┃ ‧ ┃ steps/sql/x.sql   
┃ ‧ ┃ steps/x.R ┏━━━━━━━━━━━━━━━━━━┓   
┃ sbi.Process( ┃ ┏━━━━━━━━━━━━━━━━━━┓ ┃ INSERT INTO B ┃   
┃ input = A, ┃ loads ┃ ‧ ┃ loads ┃ ‧ ┃  
┃ process = x.R, ○─────────>┃ sbi.LoadSQL(x) ○─────────>┃ FROM A ┃   
┃ output = B ) ┃ ┃ ‧ ┃ ┃ ‧ ┃  
┃ ┃ ┗━━━━━━━━━━━━━━━━━━┛ ┗━━━━━━━━━━━━━━━━━━┛  
┃ ┃   
┃ sbi.Process( ┃ steps/y.R steps/sql/y.sql   
┃ input = B, ┃ loads ┏━━━━━━━━━━━━━━━━━━┓ ┏━━━━━━━━━━━━━━━━━━┓  
┃ process = y.R, ○─────────>┃ ‧ ┃ loads ┃ INSERT INTO C ┃  
┃ output = C ) ┃ ┃ sbi.LoadSQL(y) ○─────────>┃ ‧ ┃  
┃ ‧ ┃ ┃ ‧ ┃ ┃ FROM B ┃  
┃ ‧ ┃ ┗━━━━━━━━━━━━━━━━━━┛ ┃ ‧ ┃   
┃ ‧ ┃ ┗━━━━━━━━━━━━━━━━━━┛  
┗━━━━━━━━━━━━━━━━━━┛

{  
 "firstName": "John",  
 "lastName": "Smith",  
 "age": 25  
}

## The main control script

What xsbi-core-pipeline does: - Load the sbi R package that contains custom functions for SBI-related work. - Establish the Presto connection - Initialises the pipeline (sbi.ProcessStart()) - Executes the processing steps in a series of calls to sbi.Process(), which specify the name of the input and output tables and the name of the script containing the code for the step. sbi.Process() (1) records the names of these tables and the process for the purpose of charting them as nodes in a flowchart at the end of processing, (2) loads and runs (if not a dryrun) the specified script. - Conclude the processing (sbi.ProcessEnd()). Currently, this does nothing else other than charts the flow diagram based on the data collected by sbi.Process().

## Processing steps

Currently all current processing steps are written in R but Python scripts could also be called and executed as part of the pipeline. In terms of functionality, currently most steps don’t involve more than just simply wrapping the SQL query plus some housekeeping. Most of them do the following three things: 1. Create the output table if it does not exist in Presto yet (The name of the output table is not hardc-oded into the scripts, it is a ‘global’ parameter set by sbi.Process() before it loads and executes the snippet.) 2. Wipe out the contents of the output table. 3. Load and send the SQL query to Presto. With the exception of the final aggregations, all steps involve SQL with INSERT INTO statements, so no data is pulled into memory. Final aggregations result in compact datasets (time series) which are currently held in memory as a terminal step. These will be written to an XSBI Stats Database, once that goes online.

## SQL

Presto provides a decent SQL interface for manipulating our data.[[1]](#footnote-34) All our current transformations (joining, subsetting, mutating, aggregation) in the ‘core’ XSBI pipeline can easily be written up as SQL queries, which is quite convenient. All these SQL snippets are parameterised with values injected from R, so the queries cannot be run on their own (Unless you manually inject parameter values). Parameters typically include: - The date of running the query (qureyRunDate) - Start and and dates limiting the temporal scope of the query (startDate, endDate) - Presto database (~schema) names (sbiEnv$corePath, sbiEnv$auxPath for source data tables landed by DS, sbiEnv$tempSchema for all intermediate tables the pipeline creates) - Input table names (sbiEnv$inputTable, or sbiEnv$inputTable[i] if multiple tables). These appear after FROM or JOIN. - Output table name (sbiEnv$outputTable), which appears after INSERT INTO

First Term

This is the definition of the first term. Second Term

This is one definition of the second term.

This is another definition of the second term.

1. Inlines notes are easier to write, since you don’t have to pick an identifier and move down to type the note. [↑](#footnote-ref-34)