OOS RawScans Trigger Copy to BaseScanDetail

Author: Tom Lux, Sept 2022

Contents

[Solution Summary, Background, and Purpose 1](#_Toc115248622)

[Key Notes 2](#_Toc115248623)

[Business Partners 2](#_Toc115248624)

[Requirements & Process Notes 2](#_Toc115248625)

[Code location 3](#_Toc115248626)

[Component list 3](#_Toc115248627)

[RawScans Table Trigger 4](#_Toc115248628)

[BaseScanDetail Table 4](#_Toc115248629)

[Deploy method 4](#_Toc115248630)

[Roll Back/Disable Method 4](#_Toc115248631)

[Execution Method 5](#_Toc115248632)

[Inputs 5](#_Toc115248633)

[Outputs 5](#_Toc115248634)

[Logging 5](#_Toc115248635)

[App and AppConfig Tables 6](#_Toc115248636)

[Support 6](#_Toc115248637)

[Coming Soon: 6](#_Toc115248638)

[Table Data Maintenance Job 6](#_Toc115248639)

# Solution Summary, Background, and Purpose

The new facility described herein provides an alternate table (same SQL instance and inside same database) from which perishable and non-perishable teams can extract “out of stock” data for appropriate reporting.

The name we should use to refer to this solution and the new reporting table it provides for the business is the “base scan” facility/solution/integration, because this is similar to the new reporting table (primary deliverable).

The purpose of this solution is to solve the delay in providing out-of-stock data to business teams for their reporting; the problem is further covered in INC1370380.

The primary existing/prior process (before solution herein) is driven by the “OOS Scan Processor” app, which runs as a Windows Service and processes OOS scans (made by TMs in stores) from the RawScans table (in OOS DB) into the primary reporting table (for business users) “REPORT\_DETAIL” (there is header table as well). We were not able to find source code in Azure repo that matched the log output for processor app in Prod, so we’re assuming we do not have the source code for the scan processor. In general, the OOS Scan Processor is slow, especially for weekly scans on Sundays that non-perishables team perform. Over 200,000 items are typically scanned on Sundays, and the processor does not complete population of the data into reporting tables until afternoon on Monday, so 10-20 hours later. We were not able to confirm exact cause of slowness, but we are certain that how the processor app is designed and how it queries VIM and Stella DBs for each item contributes to slow performance. Additionally, the total time increased in July for unknown reasons, which is why the referenced incident was filed. As such, this solution bypasses the heavy lookups for each item and provides the core data that our business partners need, then they lookup item info and movement separately. Furthermore, this solution runs in parallel with the existing OOS Scan Processor so business teams can confirm viability. At some point in the future, we should confirm nothing else is using the reporting tables (or any tables populated by scan processor) and that service should be stopped.

# Key Notes

1. 9/28/22: The data-purge process is not included with the initial deployment, as that would delay assessment and acceptance by our business partners.

# Business Partners

Alejandro Avellana and Akshay Mahesh represent the non-perishables team. They query the OOS DB in Alteryx workflows managed by Alejandro.

Lauren McKinley represents the perishables team (would like to add other analysts in the future). Lauren is primary contact and Tim LaPietra is alternate.

# Requirements & Process Notes

1. Requirements from Alejandro Avellana (non-perishables, weekly full scans) and Lauren McKinley (perishables, daily scans) are to provide these values:
   1. Business Unit Number
   2. Scanned-On Date+Time
   3. UPC
2. 9/19/22: Confirmed with Alejandro Avellana his team only needs scanned date, PS BU, and UPC.
3. 9/20/22: Confirmed with Alejandro Avellana that they exclude dup UPCs in a batch, so they treat a TM scanning same item twice as just 1 OOS; so, unique constraint can stand.
4. 9/21/22: Confirmed with Lauren McKinley her team doesn’t need any other fields than the three Alejandro called out.
5. Sometimes TMs scan a dummy UPC to start their scan
6. Stores scan a special UPC if they have no OOS as this gives them credit (shows they performed their OOS walk)
7. NOF or invalid items are removed by consuming teams, so no special logic was added to assess or remove these conditions.
8. Business users do not need more than a few weeks of history in the Base-Scan table, as they store OOS scan data elsewhere and do not currently have historical reporting against OOS-DB tables.

# Code location

Link to “tables” folder in Azure repo is provided below; functions, procs, and pop-data scripts are nearby and easy to find.

NOTE: No guarantees or assumptions should be made for ***any other OOS code*** in this linked repo, as the version of that code has not been verified; any of the individual components herein could be validated/verified separately.

OOS DB Code: <https://dev.azure.com/wholefoods/Supply%20Chain%20Application%20Development/_git/SCAD?path=/OutOfStock/DEV/OOSDatabase/dbo/Tables/>

# Component list

All the components herein are contained in the OOS DB (or the SQL Server level, in case of a SQL Agent scheduled job).

The core components of this facility are:

1. BaseScanDetail table
   1. Stores simplified OOS scan data; each row contains: store, scan date, UPC (plus a few other values)
   2. Each batch or list of UPCs scanned in an OOS scan by a store Team Member will have the exact same store (name and BU#) and scan date, so you can group by these fields to find/identify each scan and all UPCs it contained.
   3. This table has one or more indexes; see code for details.
2. TR\_RawScan\_AfterInsert\_CopyToBaseScanTbl trigger on RawScans table
   1. Processes scans being saved to RawScans into BaseScanDetail table, effectively splitting out the list of UPCs contained in the raw-scan JSON data.

Additional components that can be used by other apps or processes inside the OOS DB are:

1. App table
   1. Represents an app or process that can have app settings or write log entries.
2. AppConfig table
   1. Represents a runtime setting for a specific app that can be used by apps to control behavior.
3. AppLog table
   1. Represents a log entry for an app.
   2. This table has one or more indexes; see code for details.
4. WriteAppLog stored procedure
   1. Facilitates writing optional and required fields to AppLog table
5. fnGetUnknownAppID function
   1. Returns “unknown” app, if it is stored in App table, in case App references are bad/invalid.

# RawScans Table Trigger

1. The core logic is housed in this trigger.
2. This trigger fires when a row is inserted into the RawScans table, meaning when a store TM saves an OOS scan they’ve just created on a handheld/mobile device.
3. The intent of the logic in this trigger is to take each raw OOS scan, create one row for each UPC captured in the scan, and write that data to the BaseScanDetail table.

\*See trigger code in the RawScans table-definition file for details (code repo linked herein).

# BaseScanDetail Table

1. This is the alternate table to the current report\_detail table (populated by OOS Scan Processor) from which business users will extract data for out-of-stock reporting.
2. Contains:
   1. Row ID
   2. Region
   3. Store
   4. PS BU#
   5. Offset-corrected scan date
   6. UPC
   7. Insert date
3. A trigger on the RawScans table writes to this table
4. Each row inserted into RawScans (an OOS scan) is split into one row per UPC in this table
5. Data in this table will be purged by a SQL-Agent job and based on settings in the AppConfig table. See table data maintenance section for more details.

# Analysis & Assessment

Some performance metrics are saved in the trigger’s code file, so go there for more details.

One concern with this solution is “what if the trigger on the RawScans table slows or blocks OOS scans being saved by TMs in stores?”. The likelihood of this is very low, based on performance testing and real volume data from Production. The table below shows the heaviest hours during September, 2022, where the most raw scans were saved to the target table. These data show the volume of rows being saved to the RawScans table is less than 1 every 2 seconds and the non-Prod performance testing showed an average of 123 scans processed per second for 6 days worth of scans (almost 25K rows from RawScans table from Prod processed in QA). Because Prod systems are typical faster than non-prod, we should expect even better performance in Prod and therefore do not anticipate any slowness to be recognized by users creating OOS scan batches.

***OOS Production – Top 10 Scans/Min for Sept 2022***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| yr | mnth | dy | hr | totalScans | scansPerMin |
| 2022 | 9 | 4 | 18 | 1293 | 21 |
| 2022 | 9 | 25 | 18 | 1279 | 21 |
| 2022 | 9 | 18 | 18 | 1257 | 20 |
| 2022 | 9 | 11 | 18 | 1253 | 20 |
| 2022 | 9 | 18 | 19 | 1184 | 19 |
| 2022 | 9 | 25 | 19 | 1130 | 18 |
| 2022 | 9 | 4 | 19 | 1085 | 18 |
| 2022 | 9 | 11 | 19 | 1018 | 16 |
| 2022 | 9 | 18 | 17 | 947 | 15 |
| 2022 | 9 | 11 | 17 | 945 | 15 |

**\*query for above**: select yr=DATEPART(year, createdon), mnth=DATEPART(month, createdon), dy=DATEPART(day, createdon), hr=DATEPART(hour, createdon), totalScans=count(\*), scansPerMin=count(\*)/60 from rawscans (nolock) where DATEPART(year, createdon)=2022 and DATEPART(month, createdon)=9 group by DATEPART(year, createdon), DATEPART(month, createdon), DATEPART(day, createdon), DATEPART(hour, createdon) order by count(\*) desc

# Deploy method

A manual SQL script must be created from the DB components, as they have not been tested from the OOS DB project (we do not have the branch of OOS code currently running in Prod and are not spending the time to analyze or synchronize the DB project to deploy these components; it is much more efficient to simply manually deploy them from a simple SQL script). A SQL Server DB project can be used to deploy updates to a DB system, but that was not viable for the base-scan updates.

# Roll Back/Disable Method

To stop the Base-Scan facility, disable (or more drastically drop the trigger

# Execution Method

The Base-Scan facility is triggered when a row is inserted into the RawScans table (it is a table trigger in the OOS database). There is no scheduled or manual way to run or execute the process; we would need to move the logic into a stored procedure to be able to call it separately. As such, we will not attempt to populate previous days or weeks into the new base-scan table because it would require duplication of every scan in the RawScans table (we would have to insert the rows into the RawScans table, which allows exact duplicates of rows).

# Inputs

1. Row inserted into the RawScans table
2. Store table
   1. Retrieves business unit number
3. Region table
   1. Retrieves time-zone offset
4. App and AppConfig tables
   1. Retrieves settings (for core trigger app) to determine if catch-block logic is active or should be skipped

# Outputs

1. Rows inserted into BaseScanDetail table
2. Error and warning log entries written to AppLog table

# Logging

Successful, info, or debug logging is not provided by the Base-Scan facility, as it would not like provide benefit and could result in the AppLog table filling quickly.

This process only logs the following types of messages:

1. Error insert rows into the BaseScanDetail table from an OOS scan, such as if the scan is a duplicate (any of the UPCs have same scan date for that store), or any other error that could occur during insert.
2. Warning if multi-row insert is detected. The logic in the trigger does not perfectly handle more than single-row inserts at a time because during a multi-row insert, if any of the OOS-scan rows cause an error, the entire set of scans will be lost/skipped (but no data is lost or dropped from the parent RawScans table).
3. The details of the RawScans row that caused an error during copy to new base-scan table are not written to the log message, however the row ID is captured, so we can go find and review the raw data for as long as it is in RawScans.
4. The WriteAppLog proc will use a default app ID for logging, if the passed name/reference is not found in the App table.
5. The fnGetUnknownAppID function returns the first app ID for app with name containing “unknown”, or app ID ‘1’ is assumed/used (if the first identity in the App table is not ‘1’, this will cause errors).

# App and AppConfig Tables

1. The new App table is used to differentiate distinct processes or “apps”, which each have a row in this table
2. The AppConfig table then provides a means to define runtime settings for each app.

# Support

1. Check the log first, to see if that reveals immediate clues:
   1. select top 10 a.appname, al.\* from app a join applog al on a.AppID=al.AppID where al.AppID = (select appid from app where appname='Raw Scan Copy To Base Scan') order by applogid desc
2. Check last few entries in RawScans table to see if they are flowing into BaseScanDetail:
   1. select top 100 \* from BaseScanDetail order by BaseScanDetailId desc
   2. select top 50 \* from RawScans order by id desc
3. Can you confirm the table trigger firing as expected (when rows are inserted into RawScans table)?
4. Ensure setting in AppConfig table is enabled for controlling when errors and warnings are logged because if this is disabled and things are failing inside the trigger, it will not write the messages to the AppLog table.
5. Manually work through correlating data between table:
   1. RawScans (where data starts)
   2. BaseScanDetail (where trigger should be copying scan details to)
   3. Report\_Detail (table where OOS Scan Processor builds full item and movement details from each scan).

# Coming Soon:

1. Table purge process

# Table Data Maintenance Job

1. This job purges data from the following tables:
   1. AppLog
   2. BaseScanDetail
2. The list of tables that can be purged is not yet configurable, so this job only covers the new tables added during the base-scan updates (or tables we explicitly add/hard-code).
3. Data purges can be limited by either or both of these options:
   1. Total number of data rows
   2. Total number of days to retain data