

CoRS History Redesign

Late 2017

Status: Released for Approval

Version: 1.00

Target Release: 2017

**Prepared by:** Louis Puskas

**Date Created**: 10/25/2017

**Date Last Modified**: 10/025/2017

**Approvals**

Documented approvals are **required** from all approvers designated below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Project Role** | | **Approval Documentation** |
| **Mark Horn** | | **CoRS – Technology Manager** |  |
|  | |  |  |
|  | |  |  |
|  | |  |  |
|  | |  |  |
|  | |  |  |

**Contributors**

The listed individuals provided content to this document.

|  |  |
| --- | --- |
| **Name** | **Project Role** |
| Louis Puskas | **CoRS Ab Initio Team Member** |
|  |  |
|  |  |

**Version History**

Changes made during the creation and finalization of this document

|  |  |  |  |
| --- | --- | --- | --- |
| **Version #** | **Revision Date** | **Revised By** | **Change Descriptions** |
| V1.00 | 10/25/2017 | Louis Puskas | First draft created. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Glossary of Terms**

Terms and acronyms within this document that may be unfamiliar to readers.

|  |  |
| --- | --- |
| **Term/Acronym** | **Definition** |
| **Graph, Ab Initio Graph** | Ab Initio concept of a directed acyclic graph (DAG) program created in their Graphical Design Environment (GDE).  An Ab Initio graph is a data flow diagram that defines the various processing stages (or components) of a task and the streams of data as they move from one stage to another. By visually capturing the flow of information through your application, a graph lets you think in terms of meaningful processing steps instead of lines of code.  Within the framework of a graph, you can arrange and rearrange the graphically represented datasets, programs, and connections and specify the way they operate. In this way, you can build a graph to perform any data manipulation you want.  In the process of building a graph, you are developing an Ab Initio application, and thus graph development is called graph programming. |
| **Pset, Ab Initio Pset** | Parameter set used which contains specific per execution instructions into an Ab Initio graph. |
| **Plan, Ab Initio Plan** | A plan represents all the interrelated elements of an Ab Initio application and consists of:  [Tasks](http://cltaidevweb.wellsfargo.com:8080/help/topic/cos_3.3.2_en-US/Whatisatask.html#Tasks), which encapsulate your graphs, programs, and scripts  [Methods,](http://cltaidevweb.wellsfargo.com:8080/help/topic/cos_3.3.2_en-US/Whatisamethod.html#Methods) which perform the actions of tasks  [Parameters](http://cltaidevweb.wellsfargo.com:8080/help/topic/cos_3.3.2_en-US/UsingparametersinConductIt.html#UsingParameters), which pass information between tasks  [Relationships](http://cltaidevweb.wellsfargo.com:8080/help/topic/cos_3.3.2_en-US/Whatisaplan.html#Relationships), which help govern the order in which tasks run |
|  |  |
|  |  |
|  |  |
|  |  |

**Table of Contents**

[1. Document Purpose 5](#_Toc496713094)

[1.1 Business Case and Objectives 5](#_Toc496713095)

[2. Project Scope & Dependencies 5](#_Toc496713096)

[2.1 In Scope 5](#_Toc496713097)

[2.2 Out of Scope 5](#_Toc496713098)

[2.3 Project Dependencies 5](#_Toc496713099)

[2.4 Project Assumptions 5](#_Toc496713100)

[3. Background For Redesign 6](#_Toc496713101)

[3.1 EDA support 6](#_Toc496713102)

[3.2 Network infrastructure 6](#_Toc496713103)

[4. Client Requirements for Redesign 6](#_Toc496713104)

[4.1 Avoid Data Re extraction from Sql Server 6](#_Toc496713105)

[4.2 Live Changes to records capturing and rules. 6](#_Toc496713106)

[4.3 Final solution will resemble the existing CoRS code framework loading dbo schema. 7](#_Toc496713107)

[5. Solution High Level Description. 7](#_Toc496713108)

[5.1 Two part design for every history table which needs to be loaded. 7](#_Toc496713109)

[5.1 Controlled environment automation requirements. 7](#_Toc496713110)

[6. Solution Detail 8](#_Toc496713111)

[6.1 Details for History Prepare, SQL preparation graph. 8](#_Toc496713112)

[6.1.1 Explanation of different Record Subsetting Options. 12](#_Toc496713113)

[6.1.2 Explanation of different Version Number Options. 14](#_Toc496713114)

[6.1.3 Explanation of Live Changes logic and scenarios. 15](#_Toc496713115)

[6.2 Details for History Load, History SQL execution graph. 17](#_Toc496713116)

[6.2.1 History Load Graph Explanation 17](#_Toc496713117)

# Document Purpose

This document has been created to describe our latest version of CoRS Ab Initio automated data loads into our Sql Server database schema named : HistSchema.

The document will describe how the code architecture decisions came about, and how the solution to these decisions were implemented.

## 

## 1.1 Business Case and Objectives

CoRS management was seeking a redesign of existing versions of CoRS HistSchema loading code because the initial attempts to load HistSchema ran poorly when deployed into EDA’s Ab Initio shared technology environment.

# Project Scope & Dependencies

## 2.1 In Scope

| **ID** | **Description** |
| --- | --- |
| 2.1.1 | All tables which have been received data loads from earlier versions of History ETL code will also be loaded by this version. |
| 2.1.2 |  |

## 2.2 Out of Scope

| **ID** | **Description** |
| --- | --- |
| 2.2.1 | Tables which were not loaded by earlier version of History ETL code will not necessarily be supported by this version in the state it has been left at 10/15/2017 |
| 2.2.2 |  |

## 

## 2.3 Project Dependencies

| ID | Project Dependencies |
| --- | --- |
| 2.3.1 |  |
| 2.3.2 |  |

## 2.4 Project Assumptions

| ID | Assumption |
| --- | --- |
| 2.4.1 |  |
| 2.4.2 |  |

# Background For Redesign

## 3.1 EDA support

Any Ab Initio code designed by the CoRS ETL team will eventually be deployed on a network of servers and databases supported by a Wells Fargo organization referred to as EDA. This network of infrastructure pieces required to run our code could best be described as our shared technology environment. We share all our resources with members of many other teams. Since there is no system of soft or hard quotas currently put in place by EDA, we need to make code decisions which take into account peak processing times on these machines. They also limited the type of database mssql\_interface we can use during our loads to api. Api is not optimal for performance. There are other options like utility and bulk load available.

## 3.2 Network infrastructure

Our shared technology environment currently consists of Ab Initio licensed Linux servers residing in Phoenix, Arizona and Sql Server licensed database servers residing near Charlotte, North Carolina. This setup is noteworthy because any data extracted from NC has to be sent across the Wells Fargo intranet to the Ab Initio servers in AZ to be processed and then sent back to NC. Our ETL team’s and dba’s have already completed a coordinated effort to observe this interstate transmission and the result of this effort were metrics outlining network i/o as the key contributor to existing slow execution times for current versions of code loading data into our HistSchema.

# Client Requirements for Redesign

## 4.1 Avoid Data Re extraction from Sql Server

Almost all the data we need to load into our HistSchema already exists on the sql server licensed database servers in North Carolina. With this in mind, a key consideration for our redesign was to avoid re extracting and reloading any unnecessary data via our intranet bottleneck across servers in different parts of the country.

## 4.2 Live Changes to records capturing and rules.

There is a very small portion of the required data to be loaded into the HistSchema which can only be obtained from the ETL servers in Phoenix. This small subset of data is referred to as Live Changes data. Its implementation will be discussed in detail below in section six.   
  
The solution address a key business requirement wherein a front end portal user can modify the contents of a record recently committed to our DBO schema while Ab Initio jobs for the corresponding deal are still running. The business has opted to have these changes overwritten by the Ab Initio load ready data before the full history of that table is written into the HistSchema.

## 4.3 Final solution will resemble the existing CoRS code framework loading dbo schema.

During our final redesign of our history loading processes we have the opportunity to create a new framework which will more closely follow correct Ab Initio coding guidelines. Some of the key concepts the new design will leverage are : the correct use of psets residing in private projects to call generic graphs residing in public projects, avoiding unnecessary sorting of data where an in memory component could be used instead, correct layout using in our graphs, and avoiding shell interpretation in our graph parameters and components as much as possible.

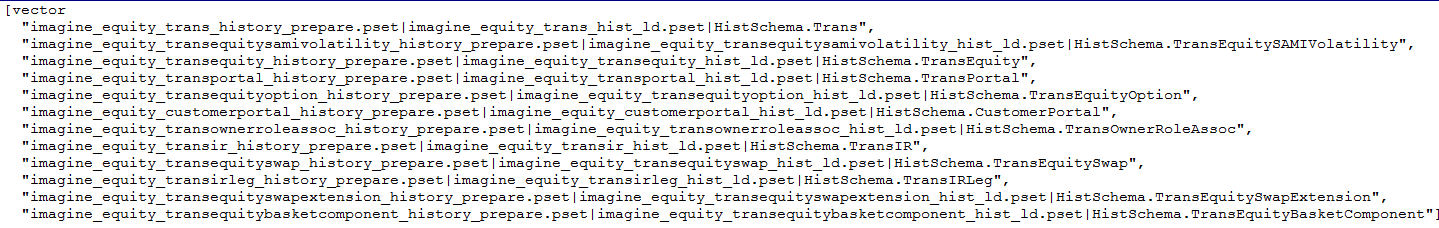
# Solution High Level Description.

## 5.1 Two part design for every history table which needs to be loaded.

Every table we will load into the Histschema will be handled by our new two part process which consists of first determining the correct sql statement to run and dynamically generating the sql statement each time, and then executing that sql statement in the existing common loading graph currently used by our Ab Initio jobs which load our dbo schema. The sql needs to remain dynamic for each execution in order to intelligently handle live changes requirements described in section 4.2 of this document.

## 5.1 Controlled environment automation requirements.

Each (two part / per table solution) described in 5.1 will be categorized into one category per SOR.   
For example one categorization named : Imagine\_Equity, will consist of the following two part jobs



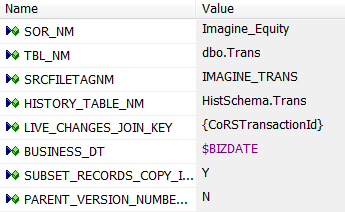
This categorization for Imagine\_Equtiy will run in what is called an Ab Initio Plan. The plan will be described in detail below in section 6.3  
The plan will be called by our Enterprise wide approved scheduler Autosys.

# Solution Detail

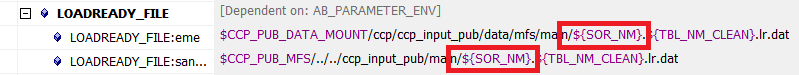
## 6.1 Details for History Prepare, SQL preparation graph.

Sql preparation in the new history framework is handled by an Ab Initio graph. The graph is located at the following location in the Ab Initio EME (source code repository). /Projects/wellsfargo/ccp/ccp\_pub/mp/ccp\_generic\_history\_prepare.mp

The graph alone cannot generate any sql required for loading history until paired with a parameter set (pset) which passes in specific instructions to the graph. The following is one such example of a parameter set which falls under the Imagine\_Equity categorization.



**SOR\_NM and TBL\_NM** : These parameters are used to uniquely identify files generate by the graph.



They are also regularly used in conditional logic throughout the graph to define rules specific to the SOR\_NM value being passed into the graph.



**SRCFILETAGNM :** This parameter is used anywhere the history prepare graph needs to add record subsetting logic into a final piece of generated sql code.

The following is an example of it being used in live change changes sql generation.

Code :



Result : The passed in parameter is highlighted yellow in the result.

select \* from dbo.Trans where SrcId in ( select SrcId

from dbo.SrcSys where SrcFileTagNm in ('IMAGINE\_TRANS'))

and updtTs > '2017-10-25 10:41:46'

and updtUserNm not like 'ccp%'

and updtUserNm <> 'CoRSMigration'

and updtTs < '20170725 19:00:00'  
and UserMchnAddr not like 'cdpra%'

**HISTORY\_TABLE\_NM :** This parameter is used to conjunction with Ab Initio’s built in sql generator utility m\_db gendml. It is also relied on in the Conduct>It plan which automates all jobs for a particular SOR\_NM.

Graph use Code : It’s figuring out what columns we’ll need to put in our sql statement.



Graph Result : BsnsDt","TransactionId","VrsnTyp","VrsnNbr","SrcId","CoRSTransactionId","BookingCustomerId"…..

Plan use Code : The use of this parameter in the Conduct>It plan is the PRIMARY reason it was left as a high level formal pset variable.

The plan parses a directory full of \*history\_prepare\* named psets in order to determine tables being loaded by the

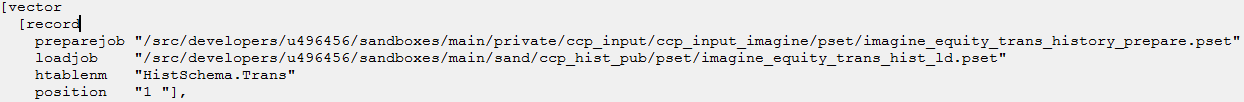
psets. Furthermore it uses this knowledge of tables being loaded per SOR\_NM to determine the correct order to load

each table.





Plan use Result :

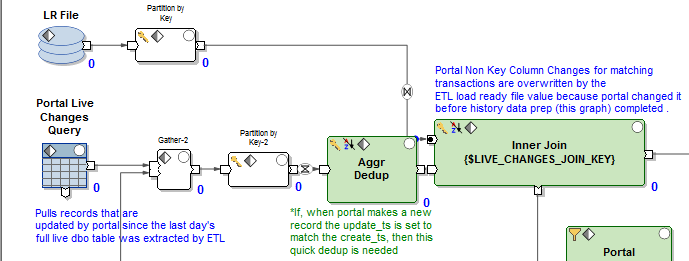


**LIVE CHANGES JOIN KEY :** This parameter is used to instruct the graph on which key to join data extracted by our live changes inquiry sql with our

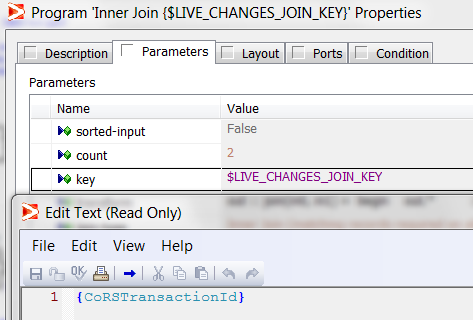
etl load ready data in case those live change need to be overwritten by the ETL load ready values. (further explained in section 6.1.3 of this

document)

Code :



Result :



**BUSINESS\_DT:** One of the key inputs to every single SQL we generate in history\_prepare is the business date the sql should be running for.

This high level graph parameter can be overridden during development to be any value the developer chooses. During an controlled environment

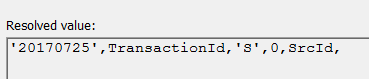
execution the value for BUSINESS\_DT will be determined by Ab Initio standard environment (stdenv) and passed into the graph dynamically.

Code :





Result :



**SUBSET\_RECORDS\_COPY\_INDICATOR:** This very important parameter has a direct impact on the final SQL generated by the history\_prepare graph by adding a clause to each sql statement which limits the number of records being handled in that particular sql statement and subsequent load graph. Please see 6.1.1 for more explanation.

**PARENT\_VERSION\_NUMBER\_VALIDATION\_INDICATOR:** This also very important parameter is used by the history prepare graph to determine which version number logic to apply to all records which have recently been copied over from the DBO schema to the HistSchema. Please see 6.1.2 for more explanation.

## 6.1.1 Explanation of different Record Subsetting Options.

As of 10/25/2017, our history framework has three supported record subsetting options.

**SUBSET\_RECORDS\_COPY\_INDICATOR Y :** A user will select ‘Y’ in this variable when they wish the generated sql to include subsetting logic. The majority of our jobs are defined this way because two or more jobs write to the same table in the Histschema. In order to keep one job from copying over data which should be copied over by, or belongs to another job, a pset which contains subset indicator ‘Y’ will leverage the value in SRCFILETAGNM to create a special clause which is append to the end of the generated sql statement.

The following is an example of a sql statement generated by a pset whose subset indicator was set as ‘Y’.

INSERT INTO HistSchema.TransPortal ("BsnsDt","TransactionId","VrsnTyp","VrsnNbr","SrcId","TradeAprvlAmt","MrkOvrdAmt","MrkOvrdExpiryDt","MrkOvrdFlg","Comment","MrkOvrdinCrncyAmt","MrkOvrdCrncyId","PortalStatId","TradeRvrslDt","CrteUserNm","CrteTs","UserMchnAddr","UpdtUserNm","UpdtTs") SELECT '20170725',TransactionId,'S',0,SrcId, TradeAprvlAmt,MrkOvrdAmt,MrkOvrdExpiryDt,MrkOvrdFlg,Comment,MrkOvrdinCrncyAmt,MrkOvrdCrncyId,PortalStatId,TradeRvrslDt,CrteUserNm,CrteTs,UserMchnAddr,UpdtUserNm,UpdtTs FROM dbo.TransPortal WHERE TransactionId NOT IN (SELECT t.TransactionId FROM dbo.TransPortal t WHERE t.TransactionId = '-99999999') AND SrcId in ( select SrcId from dbo.SrcSys where SrcFileTagNm in ('IMAGINE\_TRANS'))

The yellow highlighted section is the subset logic being added to the end of the sql statement.

**SUBSET\_RECORDS\_COPY\_INDICATOR N :**  A user will select ‘N’ in this variable when they wish the generated sql to not include any record subsetting logic. This is very useful when you have a table which needs its entire contents copied over in one bulk load. This is usually a table which independent of deals like a customer table in the dbo schema.

The following is an example of a sql statement generated by a pset who’s subset indicator was set as ‘N’.

INSERT INTO HistSchema.Customer ("BsnsDt","CustomerId","VrsnOrdrNbr","VrsnTyp","VrsnNbr","SrcId","CustomerNm","CustomerShortNm","CustomerDesc","CustomerTypId","CustomerSubtypId","CntryDomicileId","CntryCdId","RegCntryOfDomicileId……,"UpdtUserNm","UpdtTs")   
SELECT '20170725',CustomerId,VrsnOrdrNbr,'S',0,SrcId, CustomerNm,CustomerShortNm,CustomerDesc,CustomerTypId,CustomerSubtypId,CntryDomicileId,CntryCdId,RegCntryOfDomicileId,CntryOfHdqtrId,CntryOfRiskId,LglEntyIdTxt,……,UpdtTs FROM dbo.Customer WHERE CustomerId NOT IN (SELECT t.CustomerId FROM dbo.Customer t WHERE t.CustomerId = '4268578') AND CustomerId NOT IN (SELECT t.CustomerId FROM dbo.Customer t WHERE t.CustomerId = '4580957') AND CustomerId NOT IN (SELECT t.CustomerId FROM dbo.Customer t WHERE t.CustomerId = '-99999999')---------------------------------------------

The yellow highlighted section above is where the subset logic would normally be appended. However since in this example it’s turned off, there is nothing there but blanks.

**SUBSET\_RECORDS\_COPY\_INDICATOR M :**  A user will select ‘M’ in this variable when they need to use timestamps between the dbo schema and the HistSchema to determine if a record has been changed in the dbo schema since its baseline value was copied over. As of 10/15/2017, we only have two types of jobs which use ‘M’ subset settings, deal asset copies into HistSchema and deal customer portal copies into HistSchema.

The way these jobs work is the following :

1) A kick-off baseline job runs which copies over the entire content of a table to set the baseline of that table at the very beginning of processing. Example of this is : customer\_customerportal

2) After dbo jobs for a deal finish processing, the corresponding history job for that deal will kick off. Inside that deal’s history job there will be a specific job for either asset or customer. Example of this is : loaniq\_customerportal

A job like Loaniq\_Customerportal, will look into the baseline data copied into history customer portal in step one, and record an update timestamp for each customer id, as it was brought over from dbo live during the baseline copy over.

Then it will look into the current live dbo table, who’s processing just finished, and if it finds that an update timestamp greater than what it finds in the history baseline data for that id, it knows that id changed in dbo and a record of that change should be brought into history as V(existing) + 1.   
  
The following is an example SQL from a subset ‘M’ job. The key part to note for type ‘M’ is in the yellow where it’s making a look up of all the customer ID’s in customerportal which were loaded during the baseline run and their corresponding UpdtTs which was copied from the DBO schema.

with MyMaxHistTimestamp as (SELECT max(t.UpdtTs) as MaxTimeStamp, t.CustomerId FROM HistSchema.CustomerPortal t WHERE t.BsnsDt='20170725' GROUP BY t.CustomerId) INSERT INTO HistSchema.CustomerPortal ("BsnsDt","CustomerId","VrsnTyp","VrsnNbr","SrcId","RiskOfcrId","DrvtvExcsRiskOfcrFlg","FXExcsRiskOfcrFlg","BsnsCntctId1Nbr","BsnsCntctId2Nbr","AnlRvwDt","LastRvwDt","CreditVltnAmtLqdtyId……CustomerIndstId,CntryOfDomicileId,CntryOfRiskId,DeskLimitRiskOfcrId,AdtnlRptField1,AdtnlRptField3,AdtnlRptField4,SvrgnFlg,ActvFlg,AdtnlRptField2,CrteUserNm,CrteTs,UserMchnAddr,UpdtUserNm,UpdtTs FROM dbo.CustomerPortal h join MyMaxHistTimestamp m on h.CustomerId=m.CustomerId WHERE h.UpdtTs > m.MaxTimeStamp AND h. CustomerId NOT IN (SELECT t.CustomerId FROM dbo.CustomerPortal t WHERE t.CustomerId = '-99999999')  
  
The other important part of this sql to observe is the record subset logic which joins to the lookup above to determine if the record in dbo has a more recent UpdtTs than the what was earlier recorded in the baseline load.

## 6.1.2 Explanation of different Version Number Options.

As of 10/25/2017, our history framework has two supported version

**PARENT\_VERSION\_NUMBER\_VALIDATION\_INDICATOR Y :** A user will select ‘Y’ in this variable when they are building a pset to create sql intended

to load a child table who’s version number should be inherited from the parent record of the transaction. A very common example of this would be

a sql built to support the transequity table contents being copied into the Histschema. After the contents were copied over the version of each

record would be inherited from that record’s parent in the trans table.

Here is the sample sql which matches the description described above:

with MyLatestParent as (SELECT max(t.VrsnNbr) as MaxParentVersion, t.TransactionId FROM HistSchema.Trans t WHERE t.BsnsDt='20170725' GROUP BY t.TransactionId ) UPDATE HistSchema.transequity SET VrsnNbr = m.MaxParentVersion FROM HistSchema.transequity h join MyLatestParent m on h.TransactionId= m.TransactionId WHERE h.BsnsDt='20170725' AND h.VrsnNbr = 0 AND h.SrcId in ( select SrcId from dbo.SrcSys where SrcFileTagNm in ('IMAGINE\_TRANS'))

**PARENT\_VERSION\_NUMBER\_VALIDATION\_INDICATOR N :** A user will select ‘N’ in this variable when they are building a pset to create sql intended to load a parent table who is setting the master version number for that transaction. They would also select this option if they were loading a table which didn’t have any parent to child relationships to take into consideration.

Here is the sample sql which matches the description described above:

with MyLatestParent as (SELECT max(t.VrsnNbr) as MaxParentVersion, t.TransactionId FROM HistSchema.Trans t WHERE t.BsnsDt='20170725' GROUP BY t.TransactionId ) UPDATE HistSchema.trans SET VrsnNbr = m.MaxParentVersion FROM HistSchema.trans h join MyLatestParent m on h.TransactionId= m.TransactionId WHERE h.BsnsDt='20170725' AND h.VrsnNbr = 0 AND h.SrcId in ( select SrcId from dbo.SrcSys where SrcFileTagNm in ('IMAGINE\_TRANS'))

**6.1.3. Example of all possible combinations of record subsetting with version number options.**

**customer\_customer\_history\_prepare.pset**  : cid:image003.png@01D33C84.B2B25E60         
no subset, dummy max lookup

**asset\_productdetail\_assetcallputschedule\_history\_prepare.pset :**  cid:image012.png@01D33C84.B2B25E60    
no subset, max parent lookup

**imagine\_equity\_trans\_history\_prepare.pset** :  cid:image005.png@01D33C84.B2B25E60

yes subset, dummy max lookup

**asset\_tdcs\_assetrating\_history\_prepare.pset:** cid:image009.png@01D33C84.B2B25E60

yes subset, max parent lookup

**gmi\_fcm\_customerportal\_history\_prepare.pset :** cid:image015.png@01D33C84.B2B25E60

HistSchema “per id max timestamp” vs “dbo per id max timestamp” subset, dummy max lookup

## 6.1.3 Explanation of Live Changes logic and scenarios.

As was mentioned in section 6.1 of this document, there is a need to pass in a LIVE CHANGES JOIN KEY to each history\_prepare pset where the developer intends the \*history\_prepare\* job to find records which have been changed in the dbo schema while ETL processing is still ongoing, before ETL History Plans have been started.

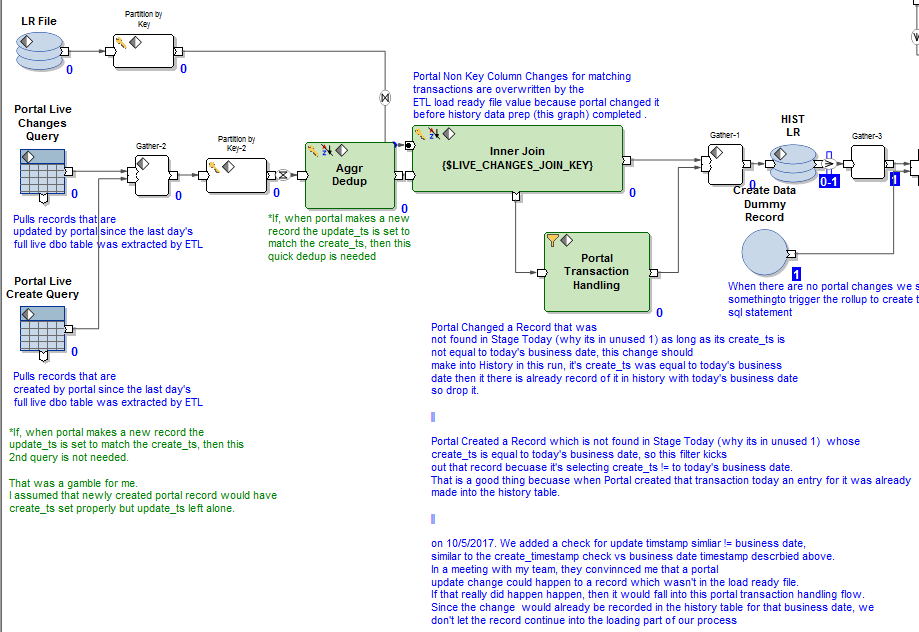
There is one very easy way to know if the history\_prepare job you’re building a pset for needs a live join key, and that is, if the table you’re working on is known to have a \*.prev.dat file created for it, within the SOR you’re working on, you will need to include a LIVE CHANGES JOIN key.

Why is this? What is so important about .prev.dat ? When you find a file a like Imagine\_Equity.TransIRLeg.prev.dat on Ab Initio Linux you know that the table has had a CDC execution run for it executed during dbo load processing. CDC cannot happen without a \*prev.dat file. The \*prev.dat file is used in the CDC execution as the baseline of all live data in a DBO table like TransIRleg which has been downloaded onto the Ab Initio server prior to the start of ETL dbo table processing.

Our team has identified this download of the all the live data in dbo tables as the absolute beginning of the LIVE CHANGES gap. They have also identified the beginning of the \*history\_preapre\* job as the absolute end of the LIVE CHANGES gap. Obviously if there is no \*prev.dat file, there is no absolute beginning to the LIVE CHANGES gap, therefore LIVE CHANGES processing is escaped in the \*history\_preapre\* job, or not done.

So what happens if a record is created or modified on the portal front end by a user who has access to our portal screens during the LIVE CHANGES gap? That record should be extracted out of the dbo schema table and its contents should be overwritten by the load ready data which was used to load the dbo schema table by Ab Initio.

The rules for this well documented in the prepare\_history code itself.



## 6.2 Details for History Load, History SQL execution graph.

So now that we’ve covered all the work involved in creating all the sql statements which need to be executed, we need to cover how the data sql statements get executed. Like mentioned before in section 5.1 of this document, each history\_prepare.pset is paired with an equivalent history\_ld.pset.

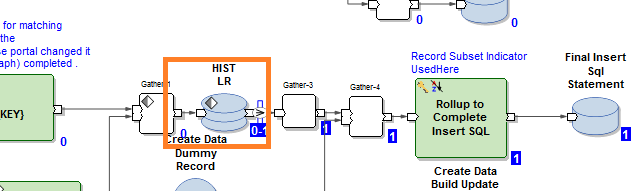
All of the history load psets can be found in the pset folder of the ccp\_hist\_pub project in the Ab Initio EME. /Projects/wellsfargo/ccp/ccp\_hist\_pub

## 6.2.1 History Load Graph Explanation

The history load jobs have two major functions which need to be executed correctly in order for the data in the HistSchema to show properly.

1. If the history prepare job found that there were LIVE CHANGES records which needed to be overwritten with Linux only available Ab Initio data, those records will need to be loaded directly from Ab Initio into SQL Server.

The prepare job will have stored those records in an intermediate file found in the history prepare graph.

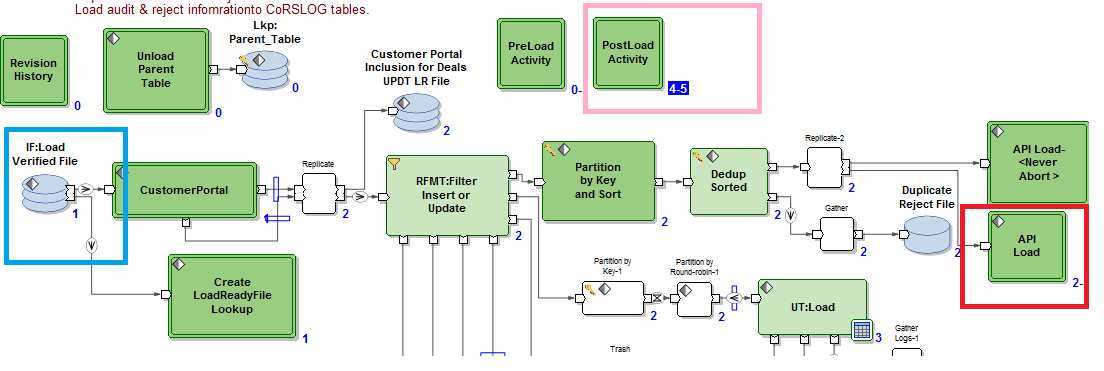


You can see the intermediate file came before the final sql statement was generated. This was done purposely because knowledge of those records needs to be included in the final sql statement.

Here is an example of this, in this sql highlighted in yellow you find the id’s of two records which will NOT be copied over using sql because they have already been inserted into the HistSchema by the loading graph before the bulk sql statements are executed. The two records highlighted in yellow below, currently reside in the load ready file in the orange box above.

INSERT INTO HistSchema.Customer ("BsnsDt","CustomerId","VrsnOrdrNbr","VrsnTyp","VrsnNbr","SrcId","CustomerNm","CustomerShortNm","CustomerDesc","CustomerTypId","CustomerSubtypId","CntryDomicileId","CntryCdId","RegCntryOfDomicileId……,"UpdtUserNm","UpdtTs")   
SELECT '20170725',CustomerId,VrsnOrdrNbr,'S',0,SrcId, CustomerNm,CustomerShortNm,CustomerDesc,CustomerTypId,CustomerSubtypId,CntryDomicileId,CntryCdId,RegCntryOfDomicileId,CntryOfHdqtrId,CntryOfRiskId,LglEntyIdTxt,……,UpdtTs FROM dbo.Customer WHERE CustomerId NOT IN (SELECT t.CustomerId FROM dbo.Customer t WHERE t.CustomerId = '4268578') AND CustomerId NOT IN (SELECT t.CustomerId FROM dbo.Customer t WHERE t.CustomerId = '4580957')

In the loading graph these two highlighted in yellow records come into the graph in the BLUE box, and are loaded into SQL SERVER in the RED box.



1. Regardless if the history prepare job found LIVE CHANGES records the load graph will need to execute whatever sql copy and version number sql statements were created in the history prepare job. This happens in the PINK box in this picture. There is a two part logic in the PINK outlined here. First we run the bulk copy sql statement, commit those changes, then we run the version setting sql logic.

