

DYNAMIC INTEGRATIONS FOR MULTIPLE HYPERION PLANNING APPLICATIONS

Giampaoli, Ricardo, TeraCorp
Radtke, Rodrigo, Dell

Abstract

In a global and competitive environment a fast access to reliable information is vital to leverage business and increase the competitive differential. Sometimes the enterprises hungering for new information that could allow them to get some advantage in a global and aggressive environment, starts to create large EPM architectures which becomes very complex over time leading to an expensive, rigid, distributed and difficult environment to maintain.

To prevent the negatives effects mentioned, this article will describe how Dell implemented a smart EPM environment that uses Oracle Data Integrator and Oracle Hyperion Planning repository to leverage its full potential, creating a centralized, reliable, responsive and extreme flexible development architecture to support the business requirements.

This was achieved with a new concept called dynamic planning integration. Using Hyperion Planning repository information it is possible to create dynamic metadata maintenance processes that changes automatically for any number of Hyperion Planning applications. It allows metadata load over any number of planning applications with low development and maintenance costs meeting the business constant need of changes.

The Journey to Dynamic Hyperion Planning ODI Integration

More and more the organizations have been investing in a global EPM environment to centralize all information in a single place giving more analytic power to the users. This is a must to have operation for all global company in the world and for Dell Inc. is not different. The growing necessity to have fast information, drove Dell to create a project to redesign its EPM architecture to a new EPM environment with a faster, more reliable and with less maintenance costs infrastructure.

The project objective was to replace the old world wide forecast application to a new one that better reflects the new direction of the enterprise and accommodate it with the existing regional applications. Analyzing the impacts that this replacement would cause in the old ODI interfaces, it was identified that the changes needed to accommodate the new application was so huge in the current infrastructure, that the creation of a multiple planning application development structure was justified.

The main challenge was the creation of a merged application that was connected with all the regional applications. The old applications were split one per region, so the key to project success was a metadata load process responsible for orchestrate all the applications, since the metadata relationship between the regional applications and the world wide one were tied.

This project also showed us how rigid and fragile is the default Hyperion Planning metadata load process using ODI for maintenance changes and new applications development. A big company cannot rely on such rigid structure that does not allows fast direction changes and new information needs. This scenario drove us not only to create new ODI interfaces to maintain this new application but also to create a new entire structure, faster, flexible, reliable and dynamic enough to support any number of new applications and changes with low development cost and time.

To have a better understanding about this new structure we will need to take a trip across the default ODI development model and see how it works behind the scenes.

Default Hyperion Planning Metadata Load using ODI: The Beginning!

Oracle Hyperion Planning is a centralized, Excel and Web-based planning, budgeting and forecasting solution that integrates financial and operational planning processes and improves business predictability. A Hyperion Planning application is based in Dimensions that basically are the data category used to organize business data for retrieval and preservation of values. Dimensions usually contain hierarchies of related members grouped within them. For example, a Year dimension often includes members for each time period, such as quarters and months.

In a Planning application, metadata means all the members in a dimension and all its properties. These properties needs to be created manually or loaded from external sources using some metadata load method. The best method to load metadata into Planning is the use of Oracle Data Integrations (ODI).

ODI is a fully unified solution for building, deploying, and managing real-time data-centric architectures in a SOA, BI, and data warehouse environment. In addition, it combines all the elements of data integration, real-time data movement, transformation, synchronization, data quality, data management, and data services to ensure that information is timely, accurate, and consistent across complex systems.

ODI uses Models and Data stores to get descriptive information about the systems and its contents. Model in ODI is a description of a relational data model and represents the structure of a number of interconnected data stores stored in a single schema on a particular technology.

Models and data store normally helps to speed up the development time, unless the target technology is Hyperion Planning and mainly if we are talking about Metadata Load. Metadata is often described as “information about data”. More precisely, metadata is the description of the data itself, its purpose, how it is used, and how the systems manages it.

Metadata load for Hyperion Planning using ODI can be simple when we are dealing with only one application and few dimensions. But it becomes extremely complex and hard to maintain if we have an environment with multiple Hyperion Planning applications with several different dimensions in each application. To exemplify this complexity let’s begin with the default metadata load process in ODI for one application that has eight dimensions and two attributes dimensions:

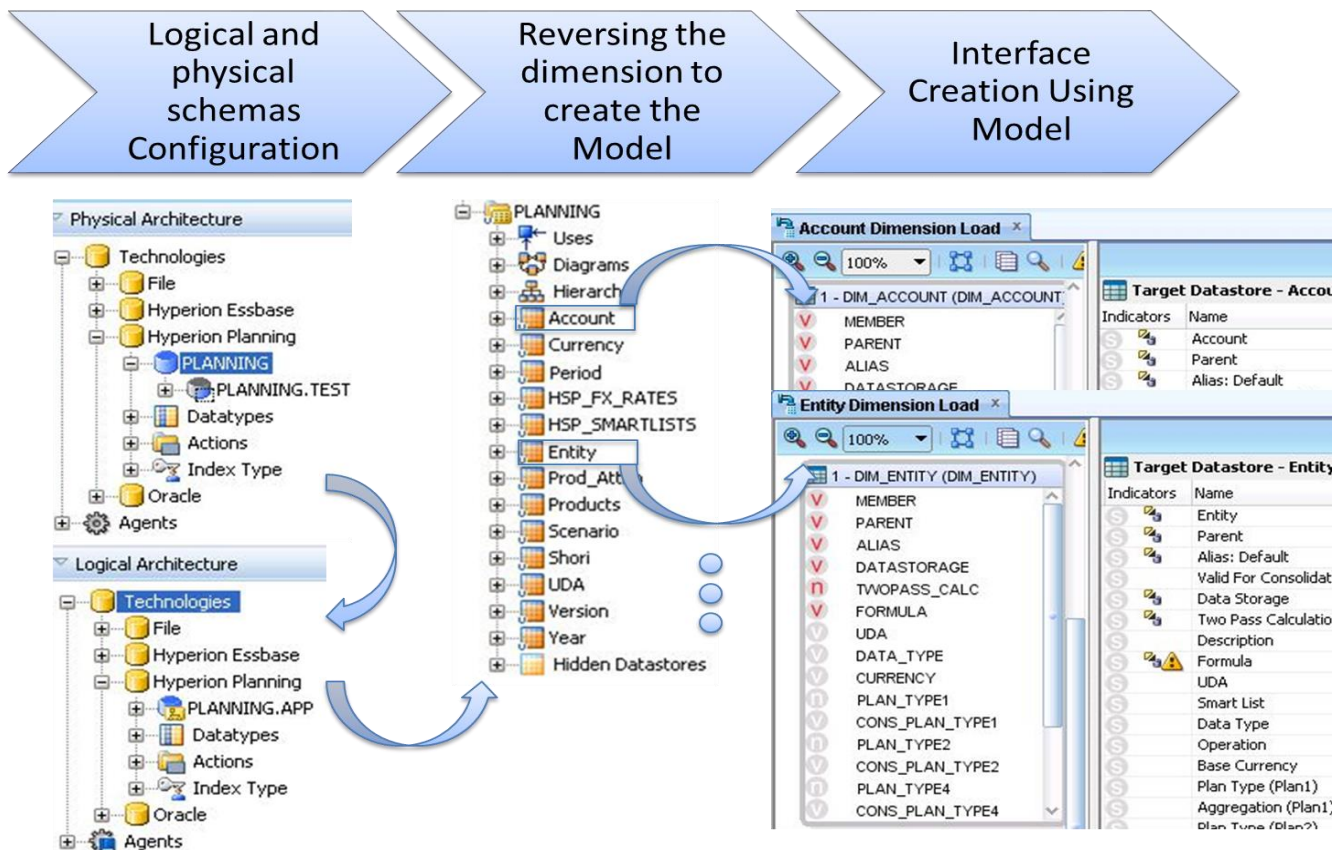


Figure 1 – Default ODI Development objects.

Figure 1 shows us how to configure a classic Hyperion planning metadata load using ODI. First step we need to setup the Hyperion Planning connection in ODI using a Logical and a Physical schema. Physical schemas are objects created in ODI that contains the connection information of a given data server. This information includes connection user/password, java driver, database URL, etc.

Logical schemas are object created in ODI that logically represents one or more physical schemas. In ODI development all references to a database object are done through a logical schema not a physical schema. This distinction between logical and physical schemas is used in ODI to differentiate what a database means in a business perspective (logical) and in a technical perspective (physical). It also allows us to use the same logical schema with different physical schemas depending on the execution context. Context in ODI is the object that bounds one logical schema to one physical schema.

Second step is to create the data store objects, a data store object in ODI represents one database object in a given technology; when we work with Oracle it means a table. In Hyperion Planning it will represent a dimension or an attribute. A dimension/attribute data store object in ODI contains all metadata information needed to be loaded into Hyperion Planning like member/parent name, consolidation type, aggregation, plan type and so on.

Third step is to create an interface object. An interface is used to build the mapping between the source data store to the target data store. It has all the information and configurations that are needed to perform our metadata load including joins, filters and data transformation.

With this we have the interface set and ready to load metadata to Planning dimensions. But what ODI does behind the scenes? This is key knowledge to understand the concept of *Dynamic* development and figure 2 explains this process.

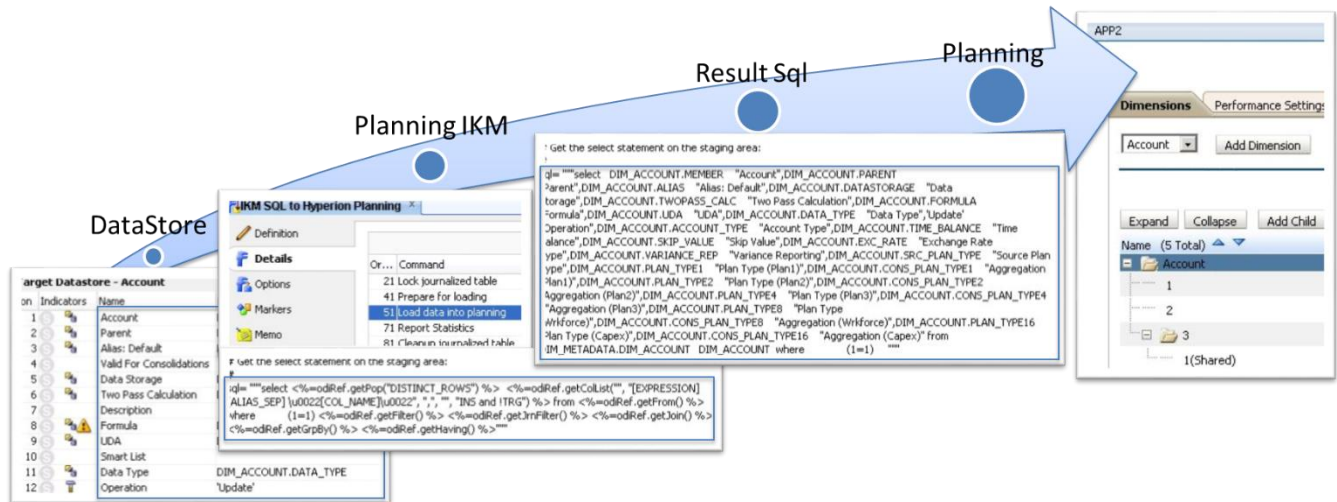


Figure 2 – Default Planning Metadata Loading process.

To perform any task in ODI we need to provide a proper Knowledge Module (KM). KM is a template containing the necessary code to implement a particular data integration task in a particular technology. These tasks include loading data, checking it for errors, or setting up triggers necessary to implement journalization. All KMs basically work the same way: ODI uses them to generate code, which then is executed by a technology at run time.

In our case ODI uses a Planning Integration KM to translate the data store information using ODI API commands to generate a SQL query that will be executed in a Java procedure to load the metadata to Hyperion Planning. This seems complicated but actually it is pretty simple.

A data store contains all necessary information to load source data to a target system. In our case this source data is an Oracle table with metadata information about account dimension as: member name, parent name, alias, data storage and so on. On the target side is the Planning data store that represents the account dimension and contains all information needed to describe a member and its functionality inside the application as two pass calc, plan type, aggregation, etc.

One important column in any planning data store is the operation column. This column accepts four different values:

- Update: Adds, updates or move the members;
- Deleted Descendants: Deletes all descendants but the member itself;
- Delete Idescendants: Deletes all descendants Including the member itself;
- Delete Level 0 members: Delete all leaf members of that particular member.

The Planning Integration KM contains a set of steps that performs different tasks needed to load metadata into planning as getting connection information, reporting load statistics and the most important one: “Load data into planning”. This step is responsible to generate the mapping code between the source and target data stores using ODI API commands. As examples "odiRef.getColList" returns all mapped columns from the target data store and "odiRef.getFrom" returns the source table name.

These commands together will generate a SQL query that will retrieve data from the source tables in the correct format that is comprehensive to Planning. With this information, ODI gathers the data and uses java to connect into Planning to load, move or delete metadata in account dimension.

And what is necessary if we want to maintain a second dimension, like Entity for example? The KM would still be the same, the source could be the same table (with different filters for example) or a completely new source table, but the target data store would definitely be different because each dimension is a separate data store in ODI so our only option is to create a new interface to load any other dimension.

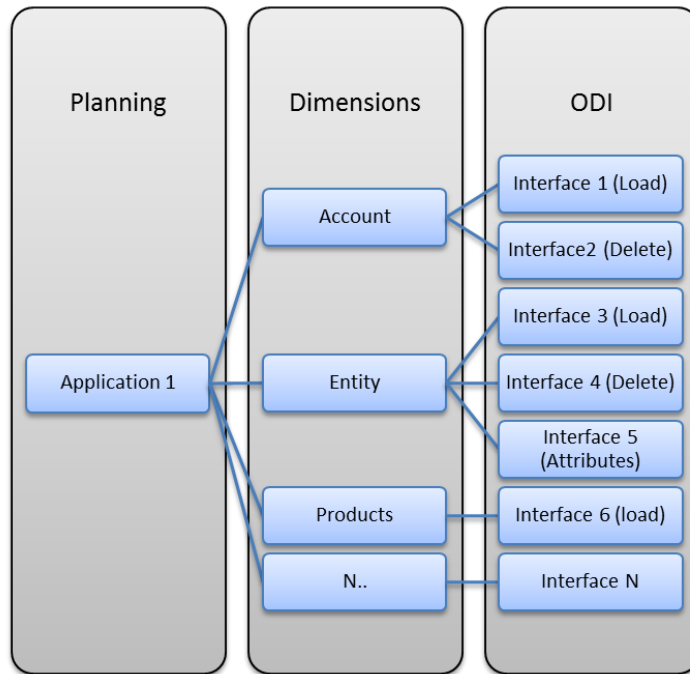


Figure 3 – Default Planning Metadata Loading overview.

Figure 3 shows what happens in a normal development environment where we end up with several interfaces per application because of the dependence between interface and dimension. Also there will be cases when more than one interface will be needed for each dimension. If we want to do something more complex like move one attribute from one parent to another, load the members in the right order or move shared members instead of creating new shared members, we will end up with two or even three interfaces per dimension. Let’s do the math: an environment with 5 Hyperion Planning applications with an average of 15 interfaces per app would give us something around 75 interfaces to code and maintain... This is a lot of interfaces, meaning bigger development time and more maintenance costs. This was the biggest motivation for us to create the concept and develop the Dynamic Integrations for Hyperion Planning Applications.

We got a problem, now what?

Develop a high number of interfaces takes a lot of time and consumes a lot of human resources. This scenario gets worse when we talk about code maintenance. If something that impacts all interfaces changes, we need to replicate the fix to all the interfaces, test all logics again and deploy them in every environment. This process could be very timing consuming and error prone. Table 1 show us the difficulties encountered in a classic development and the actions that we need to take to build a smarter and flexible solution.

Difficulties in a classic Hyperion/ODI development	Actions to achieve a smart/flexible solution
Metadata load to Hyperion Planning is limited to the creation of one interface per dimension/application generating a big volume of interfaces.	Create one generic process with the smallest number of interfaces as possible and that can be used in any number of applications and dimensions.
ODI data stores are tied to only one dimension and one	ODI data stores need to be replaced by something more flexible. This new solution needs to be independent of

application.	the application/dimension being loaded.
ODI Knowledge modules work with only one application at a time and are dependent of the target data store to know which dimension is being loaded.	ODI Knowledge models need to be upgraded to have dynamic target applications and dimensions data stores.
Metadata information generally comes from multiple sources with different data formats.	A generic metadata load process needs a standard generic inbound table for metadata. This table needs to have all the necessary columns to load metadata to Hyperion Planning and data should be in the correct format.
Each different operation that has to be done to a member in Hyperion Planning despite moving it (such as delete) requires a new separate interface do be created.	Create generic components to handle different metadata situations such as: attributes changing parents; share member load; load members in correct order and so on.
Generally, metadata load is done reading the full source tables every time to avoid problems like member order in the hierarchy and to not miss any change. This causes poor performance and may lead to a shared member creation instead of a shared member movement.	To achieve better load performance, the process needs to load only the metadata that has changed without impacting any hierarchy order or behavior.

Table 1 – Problem Solution Table.

As we can see, there are a lot of interesting and difficult points to be covered in a generic metadata load process but each of those points has a solution. Assembling all those ideas together give us a smart and flexible process that is independent of the number of applications/dimensions.

In order to achieve our goal we will need:

- A standard metadata inbound table that can be used to load any dimension and application independently;
- Another similar table to extract all information that exists in Hyperion Planning application;
- A third table to compare our inbound and extract metadata tables to create a delta between them with only the members that needs to be loaded, increasing the load performance;
- A Smart load process that understands what was changed and executes all different metadata situations like delete, move or update;
- A load component that dynamically builds its own data store information that will be used in ODI to load any application/dimension.

Sometimes it seems too good to be true but this process exists and each part of it will be explained in details in the next sessions. It all begins with having the right table structure...

Preparing to Load: Gathering the Data!

First things first! The key process in this project is to have a smart process that identify the metadata before load it into Planning. For this, we need to classify the metadata in the diverse possible categories before the load phase, creating a delta between the data coming from the diverse system of records and the Planning application itself. This delta is known as metadata tie out process. But before we can talk about this process we need to have an easy access to the new source metadata and the existing target metadata.

Inbound Process

To load any metadata into Hyperion Planning, ODI needs a set of information that describes how that member will behave inside the application, this information is specific for the dimension being loaded. For example we need to setup a “Time Balance” behavior to load an Account member and when we load a dimension that has an attribute its value needs to be loaded together with the dimension member. Each dimension has its own particularity and that is the reason why ODI needs one data store per planning dimension/application as the columns in each data store are different. Probably the source tables for each dimension will also be different, making it impossible for a generic load process to be aware of all possible inbound table sources and columns.

Account Dimension	Entity Dimension	User Defined Dimension	Attribute Dimension	Inbound Table
Account	Entity	Products	Prod_Attrib	MEMBER
Parent	Parent	Parent	Parent	PARENT
Alias: Default	Alias: Default	Alias: Default	Alias: Default	ALIAS
Operation	Operation	Operation	Operation	OPERATION
Valid For Consolidations	Valid For Consolidations	Valid For Consolidations		VALID_FOR_CONSOL
Data Storage	Data Storage	Data Storage		DATASTORAGE
Two Pass Calculation	Two Pass Calculation	Two Pass Calculation		TWOPASS_CALC
Description	Description	Description		DESCRIPTION
Formula	Formula	Formula		FORMULA
UDA	UDA	UDA		UDA
Smart List	Smart List	Smart List		SMARTLIST
Data Type	Data Type	Data Type		DATA_TYPE
Aggregation (Plan1)	Aggregation (Plan1)	Aggregation (Plan1)		CONS_PLAN_TYPE1
Aggregation (Plan2)	Aggregation (Plan2)	Aggregation (Plan2)		CONS_PLAN_TYPE2
Aggregation (Plan3)	Aggregation (Plan3)	Aggregation (Plan3)		CONS_PLAN_TYPE4
Aggregation (Wrkforce)	Aggregation (Wrkforce)	Aggregation (Wrkforce)		CONS_PLAN_TYPE8
Aggregation (Capex)	Aggregation (Capex)	Aggregation (Capex)		CONS_PLAN_TYPE16
Plan Type (Plan1)	Plan Type (Plan1)			PLAN_TYPE1
Plan Type (Plan2)	Plan Type (Plan2)			PLAN_TYPE2
Plan Type (Plan3)	Plan Type (Plan3)			PLAN_TYPE4
Plan Type (Wrkforce)	Plan Type (Wrkforce)			PLAN_TYPE8
Plan Type (Capex)	Plan Type (Capex)			PLAN_TYPE16
Account Type				ACCOUNT_TYPE
Time Balance				TIME_BALANCE
Skip Value				SKIP_VALUE
Exchange Rate Type				EXC_RATE
Variance Reporting				VARIANCE_REP
Source Plan Type				SRC_PLAN_TYPE
	Base Currency			CURRENCY
	Entity_Attrib			ATTR_ENTITY
		Product_Attrib		ATTR_PROD_ATTRIB
				APP_NAME
				DIM_TYPE
				HIER_NAME
				GENERATION
				HAS_CHILDREN

Legend
Same for all Dimensions
Same for the 3 main Types Dimensions
Exclusive of Account and Entity Dimension
Unique for each Dimension
Merge of all Possible combinations
Unique for the Inbound Table

Table 2 – Inbound Table Columns.

To standardize our metadata load process we need a standard inbound table as showed in Table 2 that can be used to load any dimension and application. Inbound table is a merge of all possible columns necessary to load all Planning dimensions of all existing Planning applications of our environment. It also contains some extra columns that are used over the generic metadata load process like:

- APP_NAME: is used to identify which app that member belongs and allows multiple applications to be loaded at the same time by the same generic process;
- HIER_NAME: indicates which dimension that member belongs, allowing multiple dimensions in one single inbound table;
- DIM_TYPE: it contains the dimension type, like ENTITY, ACCOUNT, and ATTRIBUTE DIMENSION. These are used in the generic process to decide what to do depending on the dimension type;
- POSITION: identifies which order that member should be loaded into the dimension hierarchy;
- GENERATION: contains the generation number of that member inside the dimension hierarchy;
- HAS_CHILDREN: indicates if that member has children or not.

Having this table means that we create a unique data layer where all source metadata will resides in a correct pattern that Planning understands and that the generic process reads from one standard table, not from a lot of different places.

This metadata source table can be loaded by any number of different sources such as Oracle tables, flat files, Oracle’s Data Relationship Manager (DRM) and so on. Each of these processes will require loading all necessary information about that member in a correct Planning format and they considered external processes to the generic metadata load process. Each of these external processes are unique and may contain its own business logic but the important thing is that all of them will end up populating the necessary metadata information in one generic inbound table that will be used in the generic load process. Having a central unique table for metadata also centralizes all data quality verification over the metadata itself, ensuring that only valid information goes to Planning.

Extract Process

Now that we have a standard inbound table, we will need a standard extract table that will contain all metadata that already exists in the Hyperion Planning applications and that will be compared with our inbound table information to create our delta tie out process. This table will have the same structure as the inbound table, the only difference will be the data that it will store. To populate our extract table we need to extract the existing metadata that resides in all Hyperion Planning applications but the problem is that ODI doesn't have a proper KM to extract metadata from Planning. To solve this issue a SQL query is used to extract the dimension information from the Planning Application repository itself. We could use as an alternative the ODI KM that extracts metadata from Essbase but its takes much more time and we need one interface per dimension as well.

To extract metadata from Planning we will need some SQL and Planning application repository knowledge. Before we can create a new Planning Application we need a database to store all the Planning repository tables. For all the following SQL work described in this article we will need an Oracle 11g database as we use some features only available in this version of the database. If the Planning application is not stored in an Oracle 11g database it's still possible to use what will be described here but it will be necessary to copy the data to temporary tables in an Oracle 11g database first or to adapt the code for a different version of Oracle or other different type of database.

A Planning application repository has a central table where every object that you create in the application is stored. This table is the key source to extract metadata from the applications and its name is HSP_OBJECT. From this table we can get every information needed for the extract process and it makes easy to extract any necessary dimension information. In this table we have six important columns:

- OBJECT_NAME: This column stores the object name;
- OBJECT_TYPE: This column stores the type of the member (Shared Member, Version, Entity, Time Period, Cube, Currency, Member, Planning Unit, Year, Attribute Dimension, Dimension, Alias, Folder, Form, Attribute Member, Group, User, Account, Calendar, Scenario, Currency, FX Table and User Defined Dimension Member);
- OBJECT_ID: This is the ID of the Object;
- PARENT_ID: This is the ID of the objects Parent;
- GENERATION: This informs from what generation the object belongs;
- HAS_CHILDREN: This informs if that object has a children.

We can notice that this table is based in a parent child relationship. This type of relation is perfect in cases where we want to store a hierarchy inside of a table. This way we don't need to know how many generations one hierarchy will have to create a table, we only need two columns with the parent and child IDs to rebuild that correlation. To achieve that Oracle database give us a very useful command: *Connect by Prior ... Start With*.

The *start with .. connect by* clause can be used to select data that has a hierarchical relationship and *Prior* word is used to create recursive-condition. Long story short, everything we need to do is:

```
SELECT OBJECT_NAME, OBJECT_TYPE, GENERATION, HAS_CHILDREN
FROM PLANNING_APP.HSP_OBJECT
CONNECT BY PRIOR OBJECT_ID=PARENT_ID
START WITH OBJECT_NAME='Account';
```

Dimension Name

Figure 4 – Dynamic query to extract Planning applications dimensions.

This command will retrieve all the Account Dimension hierarchy and if we need to rebuild the metadata from another dimension the only thing that we need to do is change the OBJECT_NAME to the desired dimension name, E.g. Entity. This query will be the core of our extract process. For now we have the hierarchy built, but only this is not enough as we need to have all information about the members in order to compare to our inbound table. We need to take the other metadata information from planning repository as well. Basically we need to join all the following tables together to have everything that we need:

Repository Table Name	Extract Table
HSP_OBJECT	Member
	Parent
	Generation

	Has_Children
	Postion
HSP_OBJECT_TYPE	Dim_Type
HSP_MEMBER	Data Storage
	Data Type
	Two Pass Calculation
	Aggregation (Plan1)
	Aggregation (Plan2)
	Aggregation (Plan3)
	Aggregation (Wrkforce)
	Aggregation (Capex)
HSP_ALIAS	Alias: Default
HSP_MEMBER_FORMULA	Formula
HSP_DIMENSION	Hier_Name
HSP_STRINGS	Description
HSP_ACCOUNT	Account Type
	Time Balance
	Skip Value
	Exchange Rate Type
	Variance Reporting
HSP_PLAN_TYPE	Plan Type (Plan1)
	Plan Type (Plan2)
	Plan Type (Plan3)
	Plan Type (Wrkforce)
	Plan Type (Capex)
	Source Plan Type
HSP_ENTITY	Base Currency
HSP_ENUMERATION	Smart List
HSP_MEMBER_TO_ATTRIBUTE	Associated Attributes
HSP_UDA	UDA

Table 3 – Planning Extract table mapping.

Table 3 has the mapping of the tables and the information that you can find in the application repository. Two important things to say when you build the final extraction query from planning repository:

- Remember that the HSP_OBJECT table has all the metadata information’s regarding all the objects in planning. We need to join almost all the above tables with themselves again to get the name of that particular object. E.g. HSP_ALIAS only contains the IDs to the alias, the alias itself is stored in HSP_OBJECT.

- Remember to make LEFT JOINS to all those tables against HSP_OBJECT. Depending of the dimension type we will not have anything stored in some tables. E.g. Account members doesn't have data stored in HSP_ENTITY table.

With this query ready, all we need to do is loop it passing the Dimension name to the core query mentioned above and it'll extract all dimensions from planning. Normally we learn to loop in ODI with a count variable, a check variable that checks if the loop got to the end and a procedure or a package that is called over each loop interaction. There is nothing wrong with it, but it generates more variables and a bigger flow inside the ODI package.

Thankfully we have a much easier way to create loops in ODI. In ODI we have the “*Command on Source*” and “*Command on Target*” concept. This basically enable us to execute a command in the target tab based in the command on source, that means, the command in the target will be executed for each row that returns from the query in the source tab. Basically the source query will be a cursor and the target query will be the “*Loop*” clause in an analogy to PL/SQL. Also we can pass information that returns in the source tab query to the target tab command enabling us to change the content that will be executed in the target dynamically.

With this concept we can create much simpler loops. In a procedure we can add the query that will return all dimensions that we want to loop in the “*Command on Source*”. We can get this information easily in the Application repository itself:

```
SELECT OBJECT_NAME
FROM PLANNING_APP.HSP_OBJECT HO, PLANNING_APP.HSP_DIMENSION HD
WHERE HO.OBJECT_ID=HD.DIM_ID;
```

Figure 5 –Planning application dimension.

This query will return all the dimension that exists in one planning application and with this the only thing left is to insert in the “*Command on Target*” the query to extract the data from the Planning application and then pass from the “*Command on Source*” tab the list of dimensions. To do this, basically we use the column name or the alias created in the source query as an ODI variable to the target query:

Command on Target

```
INSERT INTO METADATA_EXTRACT
SELECT OBJECT_NAME, OBJECT_TYPE, GENERATION, HAS_CHILDREN
FROM PLANNING_APP.HSP_OBJECT
CONNECT BY PRIOR OBJECT_ID=PARENT_ID
START WITH OBJECT_NAME=#DIMENSIONS;
```

Command on Source

```
SELECT OBJECT_NAME AS DIMENSIONS
FROM PLANNING_APP.HSP_OBJECT HO, PLANNING_APP.HSP_DIMENSION HD
WHERE HO.OBJECT_ID=HD.DIM_ID;
```

Figure 6 – Looping the dimensions extracting query.

This will repeat for each row returned from the source query, allowing us to extract all metadata information from all dimensions. The command on Target query on Figure 6 shows us an example of how to get some HSP_OBJECT information. To get the entire list of needed information, we use a query that joins all tables described in Table 3. It is also worth mentioning that this loop method works for every kind of looping in ODI minimizing the number of created ODI objects.

Well what a formidable deed we did, extract with only two queries all the metadata from a Planning application. But this is not enough. As the title of this articles suggests, we need to do that for any number of application, and for that we will need only to use the same loop approach again for each existing application.

Since each Planning application has its own repository we need to grant “*Select*” access to the ODI user that connects into the Oracle database to have a maximum code reutilization. With the ODI user having access to all Planning Application repositories tables all we need to do to extract all dimensions from all Planning applications is:

- Encapsulate the procedure created to extract the application dimension metadata in a ODI scenario
- Create another procedure to loop the above scenario passing the application name and the application schema in the oracle database.

How does it works? The procedure will be set with a query in the “*Command on Source*” tab that will return all the applications names that we need to loop and all the schema name for each application. This can be achieve by populating a parameter table with the name and the schema for that application or we can use the Planning repository to get this information:

```
SELECT APP.NAME AS APPS, DS.RDB_USER AS SCHEMA_APPS
FROM PLANNING.HSPSYS_DATASOURCE DS, PLANNING.HSPSYS_APPLICATION APP
WHERE 1=1
AND APP.DATASOURCE_ID=DS.DATASOURCE_ID;
```

Figure 7 – Query to get the Planning applications names and Schemas.

The Planning repository itself has some tables that stores information regarding its applications as configurations, data sources information as well the applications names and database schemas. In the HSPSYS_DATASOURCE table we can find all the information regarding the data source created to build the planning application and we also have the database schema used to setup the application repository and in the HSPSYS_APPLICATION table we can find the name of the planning applications. Join those tables together and we get all the applications names and database schemas existing in the Planning environment.

With this information available the only thing missing is to set the “Command on Target” tab with the “OdiStartScen” command. This ODI Command is used to call an ODI scenario and if we use that together with the loop approach we will be able to call that scenario as many times we need and the only code change in Figure 6 queries that we need to do are:

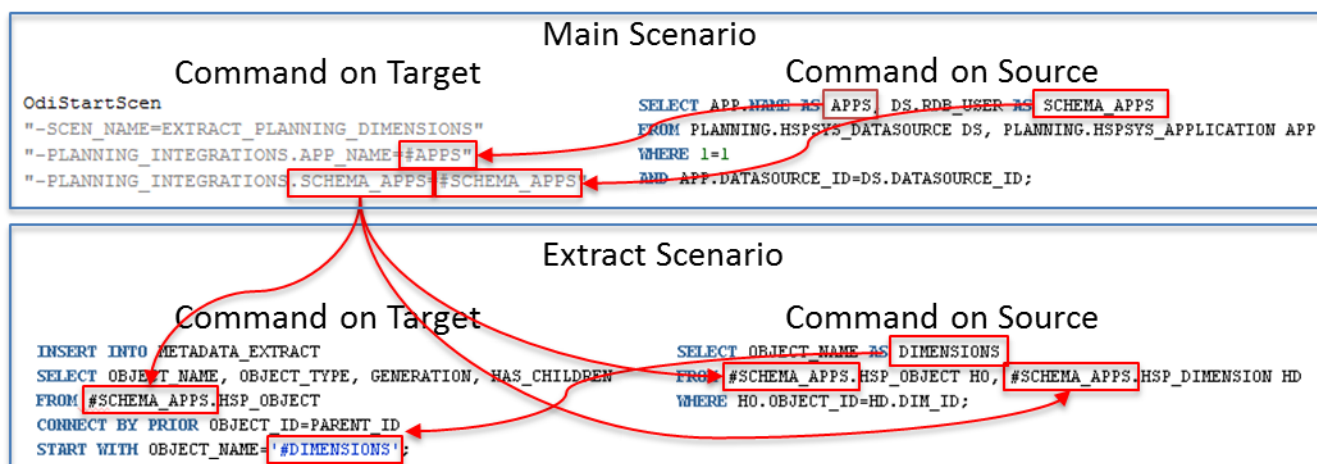


Figure 8 – Application/Dimension metadata extract Loop.

Figure 8 shows us how to extract from all Hyperion Planning applications and from all existing dimensions. This flow works as the following:

- The Main Scenario executes the first “Command on Source” query that will return all planning application names that exists in the environment together with its database schemas;
- For each line returned from the “Command on Source” an ODI scenario will be called passing as parameters the application name and the database schema to the Extract Scenario;
- Inside the Extract Scenario, it will execute the “Command on Source” query to get all existing dimensions from the inputted planning application/schema;
- For each dimension returned from the “Command on Source” an extraction query will be executed, retrieving all the necessary information to load the extract tie out table;

In the end of the process we will have the extract table loaded with all existing metadata from all planning applications and dimensions. This table will be used in the next step, where we will compare each metadata member against the source metadata and decide what to do with it.

Metadata Tie out process: More benefits than you could imagine

Now that we have an inbound and extract tables with all metadata from source and target systems we need to compare them and decide what to do with each metadata member. For this tie out process we created the metadata tie out table that is a merge of both inbound and extract tables containing all source and target columns with a prefix identifying each one of them plus a column called CONDITION. This extra column is used to describe what the metadata load process should do with that particular member. It is important for this table to have all source and target columns because then we can actually see what has changed from source to target metadata of that member.

Metadata tie out process will be responsible to read both source and extract tables and populate the metadata tie out table with all source, extract and CONDITION information. The tie out process has a built in intelligence that analyzes several different load situations to each member and populates the final result in the CONDITION column. The tie out process always searches for a parent/member/application/dimension combination in the source table and match it to the parent/member/application/dimension on the target table. The process uses this combination because these are the information that represents a unique member in Planning.

Here are the possible CONDITION statuses created by the tie out process:

CONDITION status	When it happens
Match	All metadata information from the inbound source table are equal to the extract table information, so no further action is needed.
No Match	Any column from the inbound source table is not equal to the extract table information. This member will need to be updated in the target Planning Application.
Exists only in Source	If it is a new member and exists only in the inbound source metadata table it needs to be loaded to the Planning Application.
Exists only in the Application	If a member was deleted on the source system but still remains in the planning application. For those cases we created a "Deleted Hierarchy" member and move the deleted members under it. The process don't physically delete the member to keep the data associated with it intact.
Moved Member	If a member moves from one parent to the other and needs to be updated in the Planning Application.
Changed Attribute member	When one attribute is moved from his parents to another parent.
Reorder sibling members	When a new member needs to be inserted in the place where other member previously belongs or a member changed place order with one of its siblings.
Deleted Share Members	When one shared member stops to exist in the inbound table and needs to be deleted from the Planning Application.

Table 4 – Conditions Types.

The first four conditions status are achieved by a “Full Outer Join” between the Inbound and the Extract table and a “Case When” to define the CONDITION column as we can see in the Figure 9 below:

```

INSERT INTO DIM_METADATA.METADATA_TIEOUT
(
SRC_APP_NAME,SRC_DIM_TYPE,SRC_HIER_NAME...
TRG_APP_NAME,TRG_DIM_TYPE,TRG_HIER_NAME...
CONDITION
)
(SELECT
SRC_APP_NAME,SRC_DIM_TYPE,SRC_HIER_NAME...
TRG_APP_NAME,TRG_DIM_TYPE,TRG_HIER_NAME...
CASE WHEN SRC_MEMBER = 'NA'
THEN 'Does not exist in Source'
WHEN TRG_MEMBER = 'NA'
THEN 'Does not exist in #APPS'
WHEN (
UPPER(SRC_APP_NAME)=UPPER(TRG_APP_NAME)
AND UPPER(SRC_DIM_TYPE)=UPPER(TRG_DIM_TYPE)
AND UPPER(SRC_HIER_NAME)=UPPER(TRG_HIER_NAME)
AND UPPER(SRC_PARENT)=UPPER(TRG_PARENT)
AND UPPER(SRC_MEMBER)=UPPER(TRG_MEMBER)
...
)
THEN 'Match'
ELSE 'No Match'
END CONDITION
FROM

```

- The “Case” Statement looks for ‘NA’ in the SRC_MEMBER and the TRG_MEMBER to identify data that only exists in one side, or in the Inbound either the Extract table.
- It matches all the columns that we want to check if they are equal in both sides.
- All the other data are a “No Match”

All the columns for the Inbound and Extract tables get a “SRC” and “TRG” prefix to identify each one in the Tieout table. The NVL is only to make the null fields more understandable for the Reader

```

(SELECT
NVL(SRC.APP_NAME,'NA') AS SRC_APP_NAME,NVL(SRC.DIM_TYPE,'NA') AS SRC_DIM_TYPE,NVL(SRC.HIER_NAME,'NA') AS SRC_HIER_NAME...
NVL(TRG.APP_NAME,'NA') AS TRG_APP_NAME,NVL(TRG.DIM_TYPE,'NA') AS TRG_DIM_TYPE,NVL(TRG.HIER_NAME,'NA') AS TRG_HIER_NAME...
FROM (
(SELECT SRC.APP_NAME,SRC.DIM_TYPE,SRC.HIER_NAME...
FROM DIM_METADATA.METADATA_SOURCE SRC
WHERE 1=1
AND SRC.APP_NAME = '#APPS'
)
) SRC
FULL OUTER JOIN
(SELECT TRG.APP_NAME,TRG.DIM_TYPE,TRG.HIER_NAME...
FROM DIM_METADATA.METADATA_EXTRACT TRG
WHERE 1 = 1
AND TRG.APP_NAME = '#APPS'
)
) TRG
ON ( UPPER(SRC.APP_NAME) = UPPER(TRG.APP_NAME)
AND UPPER(SRC.HIER_NAME) = UPPER(TRG.HIER_NAME)
AND UPPER(SRC.PARENT) = UPPER(TRG.PARENT)
AND UPPER(SRC.MEMBER) = UPPER(TRG.MEMBER) )
)
)

```

Full Outer Join to get all the data that exists in both sides even if they not match

Full Outer Join Clause to match by the Planning Unique key: Parent, Member and Hierarchy. In our case we need to use the Application name to since we can have more than one Application in those table.

Figure 9 – Tieout Query.

This query compares all metadata columns in the source and extract tables to see what has changed and adds to the CONDITION column what the load process should do with that row afterwards. For the other four conditions status we need to work in the data just created by the figure 9 query.

- **Moved Members:** When we execute the query from Figure 9 we get an unexpected behavior regarding moved members. A moved member is a member that changed from one parent to another. Since the query compares the member and parent names to decide if that is a new, modified or deleted member, it will consider that the source member is a new member (because it has a new parent) and the extracted member will be considered as a deleted member (because its parent/member combination does not exist in the source) generating two rows in the tie out table instead of one. To solve this issue the tie out process merge those two rows into a single one. This merge happens for all multiple rows that have the same member name but one with “Existing only in Source” condition and another one with “Exists only in the Application” condition;
- **Changed Attribute Member:** Attribute members require a special logic because Hyperion Planning treats them differently. When you want to move an attribute member from one parent to another, you first need to delete the member and then insert it back in the new parent. So this is a two-step operation, instead of the normal move member operation. When the process deletes the attribute first Hyperion Planning automatically removes its value

from its associated dimension member. If we don't load the associated dimension members again their attribute values will be missing in the end of the metadata load process. To solve this issue the metadata tie out process searches for all dimension members that have a moved attribute associated with it and change their condition to NO_MATCH. This will guarantee that after moving the attribute to a new parent the process also loads all the dimension members again with its attribute values. Another particularity with attributes is that if an attribute doesn't exist anymore in the source system it is deleted from the planning application. It is not moved to a deleted hierarchy because no data is associated directly with the attribute member, thus no data is lost;

- **Reorder sibling members:** When a single member is added to an existing parent member and this parent member has other child members, planning adds the new member in the end of the list. This is because Hyperion planning doesn't have enough information to know in which order to insert this new member as it does not have its sibling's orders to compare to it. So the tie out process also search for all existing siblings of the new member and mark them as NO_MATCH to indicate that they should be loaded all together. This way Hyperion Planning will have all siblings orders and will load the members in the correct order;
- **Deleted Share Members:** If a share member ceases to exist in the source metadata, it is removed completely from the planning application. There is no reason to move them to a deleted hierarchy member because no data is associated directly with it;

When the tie out process finishes populating the metadata tie out table we will have all information to load only the necessary members to Planning. As this table is centralized and has all applications and dimensions in it, it is just a matter to loop it for every application and dimension needed to be loaded by the generic load component. To accomplish this we will need to do some tweaking in the ODI KMs and procedures to make things more generic.

Loading a Dynamic Application

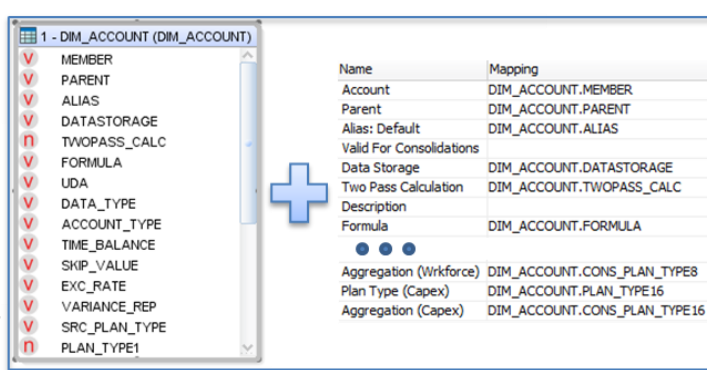
In order to create a process that is able to load any application and dimension using one single ODI interface we need to make some code changes to the KM that is responsible to load metadata into Hyperion Planning. But first we need to understand the ODI concept of a KM. KM is a set of instructions that will take the information from what exists in the source and target data stores of an ODI interface and construct a SQL command based in those data stores. In a nutshell the ODI KM is code generator based in the information that you set in the interfaces, data stores, topology and so on.

As we know the default Hyperion Integration KM is able to load only one application and dimension at a time because of the need of a target data store for each dimension in each application. If we take a deeper look in the KM to see what it does behind the scenes we will see something like this:

```

sql= """select <%=odiRef.getPop("DISTINCT_ROWS") %>
<%=odiRef.getCollist("", "[EXPRESSION] [ALIAS_SEP] \u0022[COL_NAME]\u0022", ",", "", "INS and !TRG") %>
from <%=odiRef.getFrom() %>
where (l=1)
<%=odiRef.getFilter() %>
<%=odiRef.getJrnFilter() %>
<%=odiRef.getJoin() %>
<%=odiRef.getGrpBy() %>
<%=odiRef.getHaving() %>"""""

```



Name	Mapping
Account	DIM_ACCOUNT.MEMBER
Parent	DIM_ACCOUNT.PARENT
Alias: Default	DIM_ACCOUNT.ALIAS
Valid For Consolidations	
Data Storage	DIM_ACCOUNT.DATASTORAGE
Two Pass Calculation	DIM_ACCOUNT.TWOPASS_CALC
Description	
Formula	DIM_ACCOUNT.FORMULA
● ● ●	
Aggregation (Wrkforce)	DIM_ACCOUNT.CONS_PLAN_TYPE8
Plan Type (Capex)	DIM_ACCOUNT.PLAN_TYPE16
Aggregation (Capex)	DIM_ACCOUNT.CONS_PLAN_TYPE16

```

sql= """select DIM_ACCOUNT.MEMBER "Account",
DIM_ACCOUNT.PARENT "Parent",
DIM_ACCOUNT.ALIAS "Alias: Default",
DIM_ACCOUNT.DATASTORAGE "Data Storage",
DIM_ACCOUNT.TWOPASS_CALC "Two Pass Calculation",
DIM_ACCOUNT.FORMULA "Formula",
...
DIM_ACCOUNT.CONS_PLAN_TYPE8 "Aggregation (Wrkforce)",
DIM_ACCOUNT.PLAN_TYPE16 "Plan Type (Capex)",
DIM_ACCOUNT.CONS_PLAN_TYPE16 "Aggregation (Capex)"
from STAR_SCHEMA.DIM_ACCOUNT DIM_ACCOUNT
WHERE (l=1) """""

```

Figure 10 – KM behind the scenes.

Basically what the KM does is translate the Planning application data store to a SQL query, and as we know, we get this data store by reversing a Planning application inside ODI. Fair enough, but this also means that if we could somehow have the same information that ODI has to reverse this application dimension to a data store we could easily end up with the same SQL created from that data store information. As we already showed before we have the Planning application repository itself where all the information about a Hyperion application is stored. We only need to read this information to get the same information provided by the ODI data store.

Knowing this the only thing left is to change the default KM according to our needs, and for this we need to make three changes on it:

- Make the application name that it is going to be loaded dynamic;
- Make the dimension name that is going to be loaded dynamic;
- Change the way that the KM builds its SQL command that will load metadata to Hyperion Planning. Currently it builds its SQL command based on the source and target data stores and the interface mappings;

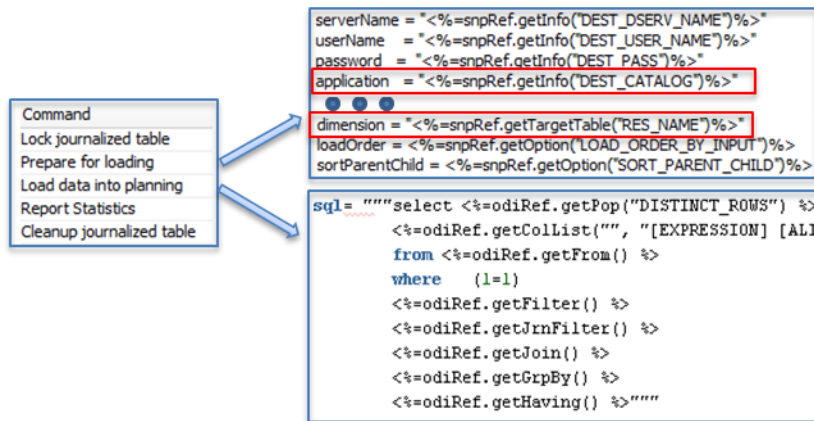


Figure 11 – Default KM behind the scenes.

In Figure 11 we can see how a default planning integration KM works. Basically it has two main steps: “Prepare for loading” and “Load data into planning”. The first one is responsible to set all information regarding connections, log paths, load options and so on. The second step is responsible to retrieve all source data based in the interface mapping and the source/target data store and load it to planning. In our case, the application and dimension names resides on the first step and the SQL command resides in the second step so we already know where we need to change the code.

But we need to analyze further to know what exactly we need to change. For the application name ODI gets it from “<%=snpRef.getInfo("DEST_CATALOG")%>” API function that returns the application name based in the destination target store that is connected to a logical schema that finally resolves into a physical schema that contains the application name itself. If we change it to an ODI variable we will be able to encapsulate this interface into an ODI package and loop it passing the application name as a parameter, making it independent of the target data store topology information and giving us the ability to load any Hyperion planning application using one single interface.

The dimension name follows the same logic: ODI gets it from “<%=snpRef.getTargetTable("RES_NAME")%>” API function that returns the resource name from the target data store that in this case is the dimension name itself. Again if we changed it to an ODI variable we will be able to encapsulate this interface into an ODI package and loop it passing the dimension name as a parameter, making it independent of the target data store resource name and enabling us to load any dimension with one interface.

The third part is the most complex one. ODI data stores for planning applications are so different from one dimension to another that they require one data store object for each dimension. In figure 10 we can see that ODI relies on “odiRef.getColList” API command to return all mappings done in the target dimension data store, which has the correct dimension format required to load that dimension metadata into planning.

So the big question is: How can we change the “Load data into planning” step to use a dynamic SQL to create dynamic interface mappings to load to any application/dimension? The answer is to rely again on the “Command on Source/Target” concept and on the planning repository metadata information.

Instead of getting the mapping information from the ODI data store object, we can query Planning repository to get the same mapping for all dimensions and applications being loaded. The result of this query is a formatted mapping, identically of what ODI would have generated if we used the default planning development, but with the big advantage of being entirely dynamic to any application and dimension.

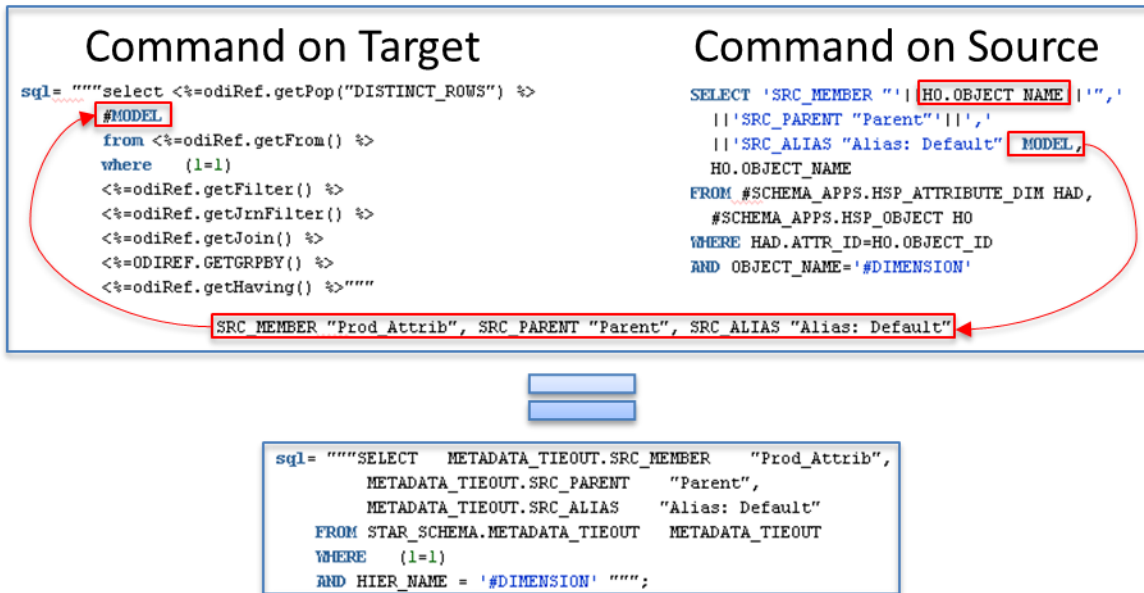


Figure 12 – Dynamic KM behind the scenes.

In figure 12 we can see an example using an Attribute dimension. The command on source will query HSP_OBJECT and HSP_ATTRIBUTE_DIM of a given application (defined by #SCHEMA_APP variable) to retrieve information about one attribute dimension (defined by #DIMENSION variable). Those variables are passed from an external ODI package that will be used to loop all applications and dimensions that we want to load.

Account Dimension	Entity Dimension	User Defined Dimension	Attribute Dimension
Account	Entity	Products	Prod_Attrib
Parent	Parent	Parent	Parent
Alias: Default	Alias: Default	Alias: Default	Alias: Default
Operation	Operation	Operation	Operation
Valid For Consolidations	Valid For Consolidations	Valid For Consolidations	
Data Storage	Data Storage	Data Storage	
Two Pass Calculation	Two Pass Calculation	Two Pass Calculation	
Description	Description	Description	
Formula	Formula	Formula	
UDA	UDA	UDA	
Smart List	Smart List	Smart List	
Data Type	Data Type	Data Type	
Aggregation (Plan1)	Aggregation (Plan1)	Aggregation (Plan1)	
Aggregation (Plan2)	Aggregation (Plan2)	Aggregation (Plan2)	
Aggregation (Plan3)	Aggregation (Plan3)	Aggregation (Plan3)	
Aggregation (Wrkforce)	Aggregation (Wrkforce)	Aggregation (Wrkforce)	
Aggregation (Capex)	Aggregation (Capex)	Aggregation (Capex)	
Plan Type (Plan1)	Plan Type (Plan1)		
Plan Type (Plan2)	Plan Type (Plan2)		
Plan Type (Plan3)	Plan Type (Plan3)		
Plan Type (Wrkforce)	Plan Type (Wrkforce)		
Plan Type (Capex)	Plan Type (Capex)		
Account Type			
Time Balance			
Skip Value			
Exchange Rate Type			
Variance Reporting			
Source Plan Type			
	Base Currency		
	Entity_Attrib		
		Product_Attrib	

Legend
Same for all Dimensions
Same for the 3 mainly Dimensions
Exclusive of Account and Entity Dimension
Unique for each Dimension

Table 5 – Dimensions Data Store information.

If we take a further look into all different data stores that a Planning application could have, we will see a pattern regarding the information that we need to send to Planning to load metadata depending of each dimension, as we can see in the Table 5.

The logic to create the dynamic mapping columns is exactly the same used to create the inbound and the extract tables. The only difference is that for the inbound and extract tables we need to put all columns together and for the KM mapping we need to, depending of the selected dimension, take the right information in the application repository. This information will help us to create the necessary mapping that contains the right source columns and the right alias of those columns, which will inform Planning about what that metadata column stands for.

Since our metadata tie out table contains standard columns for all dimensions we don't need to worry about adjustments when we change to another dimension, and since our source metadata table already has the metadata information in the correct planning format, we don't even need any kind of transformation here, it is just a matter to read from the metadata source table and load directly to Planning.

In the Figure 12 example we will use the SRC_MEMBER, SRC_PARENT and SRC_ALIAS as the mapping columns and for the Planning alias the only one that is dynamic is the member name alias that identifies the dimension name. To get this information we need to query the application repository looking for the attributes into HSP_ATTRIBUTE_DIM and for its name in HSP_OBJECT table, and finally we can use the OBJECT_NAME column to get the dimension name alias.

Executing this query we will get a one line mapping string that will be passed as a parameter (#MODEL) from "Command on Source" to "Command on Target" and will enable ODI to load metadata to that specific dimension/application. If we execute this interface and look at the query created in ODI operator we will see that the result is the same as a default KM would create but with the big advantage of being entirely dynamic. Following this logic, we would only need to change the value of the #SCHEMA_APP and #DIMENSION variables to get another application\dimension loaded.

Off course we need to work a little more to get the mapping for the other dimensions as Account or Entity, but the idea will be always the same: query the application repository to get the data store information depending on the dimension\application selected.

Tie Out Table	Planning Alias	Mapping	Type
SRC_MEMBER	Member	'SRC_MEMBER " HSP_OBJECT.OBJECT_NAME	Dynamic
SRC_PARENT	Parent	'SRC_PARENT "Parent"	Fixed
SRC_ALIAS	Alias: Default	'NVL(SRC_ALIAS,"<NONE>" "Alias: Default"	Fixed
OPERATION	Operation	From an option in IKM.	Dynamic
SRC_DATASTORAGE	Data Storage	'SRC_DATASTORAGE "Data Storage"	Fixed
SRC_TWOPASS_CALC	Two Pass Calculation	'SRC_TWOPASS_CALC "Two Pass Calculation"	Fixed
SRC_FORMULA	Formula	'SRC_FORMULA "Formula"	Fixed
SRC_UDA	UDA	'NVL(SRC_UDA,"<NONE>" "UDA"	Fixed
SRC_DATA_TYPE	Data Type	'SRC_DATA_TYPE "Data Type"	Fixed
SRC_CONS_PLAN_TYPE1	Aggregation (Plan1)	'SRC_CONS_PLAN_TYPE1 "Aggregation (HSP_PLAN_TYPE.TYPE_NAME)"	Dynamic
SRC_CONS_PLAN_TYPE2	Aggregation (Plan2)	'SRC_CONS_PLAN_TYPE2 "Aggregation (HSP_PLAN_TYPE.TYPE_NAME)"	Dynamic
SRC_CONS_PLAN_TYPE4	Aggregation (Plan3)	'SRC_CONS_PLAN_TYPE4 "Aggregation (HSP_PLAN_TYPE.TYPE_NAME)"	Dynamic
SRC_CONS_PLAN_TYPE8	Aggregation (Wrkforce)	'SRC_CONS_PLAN_TYPE8 "Aggregation (HSP_PLAN_TYPE.TYPE_NAME)"	Dynamic
SRC_CONS_PLAN_TYPE16	Aggregation	'SRC_CONS_PLAN_TYPE16 "Aggregation (Dynamic

	(Capex)	HSP_PLAN_TYPE.TYPE_NAME ')"'	
SRC_PLAN_TYPE1	Plan Type (Plan1)	'SRC_PLAN_TYPE1 "Plan Type (HSP_PLAN_TYPE.TYPE_NAME ')"'	Dynamic
SRC_PLAN_TYPE2	Plan Type (Plan2)	'SRC_PLAN_TYPE2 "Plan Type (HSP_PLAN_TYPE.TYPE_NAME ')"'	Dynamic
SRC_PLAN_TYPE4	Plan Type (Plan3)	'SRC_PLAN_TYPE4 "Plan Type (HSP_PLAN_TYPE.TYPE_NAME ')"'	Dynamic
SRC_PLAN_TYPE8	Plan Type (Wrkforce)	'SRC_PLAN_TYPE8 "Plan Type (HSP_PLAN_TYPE.TYPE_NAME ')"'	Dynamic
SRC_PLAN_TYPE16	Plan Type (Capex)	'SRC_PLAN_TYPE16 "Plan Type (HSP_PLAN_TYPE.TYPE_NAME ')"'	Dynamic
SRC_ACCOUNT_TYPE	Account Type	'SRC_ACCOUNT_TYPE "Account Type"'	Fixed
SRC_TIME_BALANCE	Time Balance	'SRC_TIME_BALANCE "Time Balance"'	Fixed
SRC_SKIP_VALUE	Skip Value	'SRC_SKIP_VALUE "Skip Value"'	Fixed
SRC_EXC_RATE	Exchange Rate Type	'SRC_EXC_RATE "Exchange Rate Type"'	Fixed
SRC_VARIANCE_REP	Variance Reporting	'SRC_VARIANCE_REP "Variance Reporting"'	Fixed
SRC_SRC_PLAN_TYPE	Source Plan Type	'SRC_SRC_PLAN_TYPE "Source Plan Type"'	Fixed
SRC_CURRENCY	Base Currency	'SRC_CURRENCY "Base Currency"'	Fixed
SRC_ATTR_ENTITY	Entity_Attrib	'SRC_ATTR_ENTITY "" HSP_OBJECT.OBJECT_NAME	Dynamic
SRC_ATTR_PROD_ATTRIB	Product_Attrib	'SRC_ATTR_PROD_ATTRIB "" HSP_OBJECT.OBJECT_NAME	Dynamic

Table 6 – Dimensions Mapping information

In table 6 we can see all the possible mapping combination that we can have in a planning application for the mainly planning dimensions and we notice that some information are dynamic (dependent of the planning repository) and some are fixed. To put everything together in one single query here are some tips:

- The majority of the columns are fixed and can be obtained with a simple *“select ‘Any string’ from dual”*;
- The easiest way to create this SQL is to create separated SQLs for each different kind of information and put everything together using *Unions* statements;
- Split the final query in small queries to get the different categories presented in table 5;
- Use the MULTI_CURRENCY column in HSP_SYSTEMCFG table to find out if that application is a multicurrency one or not;
- For aggregations and plan type mapping we need to get the name of the plan type itself and for this we use the HSP_PLAN_TYPE table;
- When the query is ready you need to add a filter clause to filter the dimension from where that information belongs;

With the query ready the only missing step is to insert it into the “Command on Source” tab inside the Planning IKM and pass the string generated by it to the “Command on Target” tab as we can see in the figure 12.

This ends all the preparations that we need for the next step that is to put everything that we have learned into an ODI package that will dynamically load metadata into any number of Planning applications.

Putting everything together: Dynamic Hyperion Planning metadata integration in action

After having explaining each component in separate it is just a matter of assembling all pieces together to have a generic process that can load any planning dimension to any application using generic components. The ODI architecture was created to be as much modular as it can be, meaning that each component is responsible for a very specific piece of code, and with one main ODI scenario that is responsible to orchestrate and call each one of those specific scenarios needed for this process. A resumed flow of this main ODI scenario looks like the diagram in figure 13.

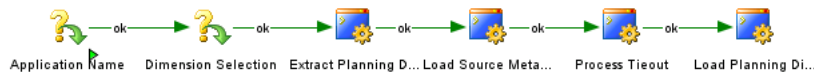


Figure 13 – Dynamic Integration Scenario.

The process accepts two input parameters. The first one is “Application Name” that indicates which application the metadata maintenance will be loaded into. In this case the user can input one application at a time or he can input “ALL”, indicating that he wants to load metadata to all applications contained in a previous populated parameter table (that will hold the valid application names) or all existing applications in the planning repository like in Figure 7.

The parameter “Dimension Selection” will indicate which dimension will be loaded to planning. Again in this case the user can select on dimension at a time or he can input “ALL”, indicating that the process will load all existing dimensions of the select application.

After the input parameters, the process is divided in big four components: Extract Planning Dimensions, Load Source Metadata, Process Tie out and Load Planning Dimensions. Each of these components may call several ODI child scenarios that will execute its process for one specific application and dimension, allowing us to load everything in parallel, giving us a huge performance gain. Let’s detail each of these components and see how they behave in the final load process.

Extract Planning dimensions

The extract process is separated in two components: the first component is a very simple procedure that resides in the ODI main package (figure 13) and is responsible to create a loop using the “Command on Source” and “Command on Target” concept with all applications that needs to be extracted. The source tab gets all applications that exist in the planning repository or in a parameter table and the command on target calls an ODI scenario that contains the second extract component for each line returned from the source tab.

The second extract component is responsible to get each application passed from the first process and extract all metadata information regarding the dimensions that exists in that planning application to the standard extract table. This second component also relies on “Command on Source” and “Command on Target” concept to get all existing dimension or just the one dimension passed as user parameter in the source tab and insert that dimension data into the extract table in the target tab. In the end of this process we will have the metadata extract table with all metadata information that exists in all planning applications/dimensions valid for that execution.

Load Source Metadata

This component is responsible to call one or more scenarios that will populate the inbound metadata table. Since the metadata may come from different kind of sources (such as Oracle tables, text, csv files and so on) this is not a fixed component and it may be altered to add or remove any call to external load scenarios. Those external load scenarios will be developed to populate the inbound metadata table and may contain a lot of requirements, rules and data transformation on them. This is the key benefit of having a standard inbound table for a metadata maintenance process: you may add any number of load process as you want/need to load only one single inbound table and after that all the load process into Planning is untouched and generic to any number of applications and dimensions, decreasing the development and maintenance cost.

Process tie out

After having both inbound and extract tables filled with metadata information, the process populates the metadata tie out table as explained in section “Metadata tie out process: more benefits than you could imagine”. This procedure does not need to call any auxiliary scenario or create any kind of loop because all information regarding all applications/dimensions are now placed in the inbound and extract tables, so it is just a matter to read all that information and populate the metadata tie out table with the correct CONDITION status. In the end of this process we will end up with a table that has all information regarding on what to do with each metadata member that needs to be loaded into the planning applications.

Load Planning dimensions

The load process is also divided in two components: the first component is a simple procedure that resides in the ODI main package (figure 13) and is responsible to create a loop using the “Command on Source” and “Command on Target” concept with all applications/dimensions that needs to be loaded. The source tab gets a distinct of all applications and dimensions that exist in the metadata tie out table filtering just the members that doesn’t have a “Match” CONDITION status. The CONDITION status is filtered here to get only those applications and dimensions that had any member that was changed or that needed to be loaded, avoiding loading any unnecessary metadata into planning and increasing the performance load times. Then the command on target calls an ODI scenario that contains the second load component for each line returned from the source tab.

The second extract component is responsible to get each application and dimension passed from the first process and loads all metadata information regarding that dimension from the metadata tie out table. This component is separated in three load steps as described below:

- **Delete Attribute members:** This step filters the metadata tie out table looking if that dimension is an attribute dimension, and if yes, it sends to Planning application all members with the CONDITION status as “No Match” or “Deleted Attribute” using the Load Operation “Deleted Idescendants”. This will delete all attribute members that changed from on parent to another (because Hyperion Planning does not change attribute members automatically) and delete all attribute members that does not exist anymore in the source metadata table;
- **Delete Shared members:** This step sends to Planning application all members with the CONDITION status as “Deleted Share” using the Load Operation “Deleted Idescendants”. This will delete all shared members that does not exist anymore in the source metadata table;
- **Load Planning members:** This step sends to Planning application all members with the CONDITION status that are NOT “Match”, “Deleted Share” or “Deleted Attribute” using the Load Operation “Update”. This will load all new and modified members, shared members and attributes. It will also move all deleted members to its respective deleted hierarchy;

And that finishes the load process. Load all metadata information from any number of Hyperion Planning applications/dimensions using only one generic component with three generic steps. It’s pretty amazing hum?

Conclusion – Dynamic Integrations in real environment

This articles show us the challenges to build a centralized and flexible development architecture for Hyperion Planning application using ODI to maintain metadata for any number of applications and dimensions and at the same time have a prepared environment to accept new applications or great changes in the current ones. All this hard work was compensated by the benefits that this new structure delivered to the business as we can see in the next table, that show us the major differences between the old and new structure.

	Old Structure	New Structure
Metadata Load Process	All metadata is loaded every time the process runs	Only the new or the changed metadata is loaded every time the process runs
Execution Time	8 hours to finish the process for all regions applications.	1 hour to finish the process for all regions applications. (depends on the amount of data that was changed)
Total Number of ODI Objects	Multiple interfaces depending on the number of Applications and dimensions Number of Packages: 19 Number of Interfaces: 51 Procedures: 32	One generic interface that load metadata regardless the amount of applications or dimensions Number of Packages: 8 Number of Interfaces: 6 Procedures: 9
Development Cost	Great effort replicating similar interfaces for new applications and	Needs only to configure the process parameter to include the new

	dimensions	application or dimension
Maintenance Cost	Multiple changing/error points to be verified each time a change is needed	Centralized code for all applications. Only one place to change if required.
Scalability	Need to create new interfaces to supply new applications	No need to create any new interface. Code automatically identifies a new application based in a configuration table.

Table 7 – Process final results.

Those are the biggest impacts that were identified after the project go live, and after that, we already had two other Planning projects: a new Planning application, and another one that changed all the entity hierarchy to reflect a new way to present the financial data.

For including a new application the development time was minimal and was restricted to some inserts into the parameter table plus a union in the procedure (external ODI package) that takes the metadata from the source to the inbound table and transform the data as planning requires. That was a great test to see if all the effort was paid off, and it certainly was, as the development phase took only four days to be completed.

For the second project that changed all the entity structure there was no development effort because the designed metadata process can handle metadata load as well metadata change or deletions, so the only action necessary was to execute the process to get the new data into the Planning application and move the old metadata to the deleted hierarchy, everything in only one single execution of the metadata load ODI scenario.

Since this project had a final delivered date already scheduled, we did the best that we could, in a limited space of time. In the middle of this project some other new ideas came up but were not implemented due the lack of time, like use the Essbase API to get information about the ASO cubes and use it in the same way that we did with the planning application repository to get its ODI data store information and include the ASO cubes to this dynamic architecture. This could be a future project to include not only Planning application to the metadata maintenance process but also any ASO cube that the company may have.

ODI is a dynamic and very powerful tool that can provide an excellent platform to create and maintain EPM environments, and this article was created to show that our imagination is the only limit to accomplish it. With a few changes in the KM and a structured set of tables we can overwhelm the boundaries of the default development, achieving a new level of excellence giving the company more flexibility, reliability and scalability to conquer the challenges of a global and competitive environment.