Designing a Data Virtualization Environment
A Step-By-Step Approach

A Technical Whitepaper

Rick F. van der Lans
Independent Business Intelligence Analyst
R20/Consultancy

January 2016

Sponsored by

Table of Contents

1 Management Summary

This whitepaper describes a *step-by-step approach* for setting up a data virtualization environment developed with the *Red Hat JBoss Data Virtualization Server* (JDV), which is based on the community project *Teiid*. It contains do's and don'ts and guidelines to help organizations develop an effective and efficient data virtualization environment.

The recommended architecture consists of minimally four layers of views:

- **Virtual Base Layer** views give access to data stored in source systems and are partly responsible for the data quality.
- **Enterprise Data Layer** views present an integrated view of all the source systems' data.
- **Shared Specifications Layer** views contain shared specifications to avoid duplicate and possibly inconsistent specifications in the data consumption views.
- **Data Consumption Layer** views simplify data access for data consumers.

A *vertical* design and development approach is recommended. With this approach a limited number of views is defined within a layer before work starts on views in the next layer. The vertical approach fits very well with current agile design techniques[1] where development is highly iterative. Also, it works well for modern-day data warehouses where the requirement is to be able to develop more quickly and to change existing reports more quickly.

The process to develop a new JDV environment or to change an existing one consists of the following steps:

1. Designing and implementing the virtual base layer
2. Designing and implementing the enterprise data layer
3. Designing and implementing the shared specifications layer
4. Designing and implementing the data consumption layer
5. Designing and implementing views for self-service usage
6. Designing and implementing views to manage data quality
7. Optimizing performance

These seven steps are not meant to be executed one by one, but should be seen as a highly iterative process.

Note: The whitepaper doesn't include a description of the installation of JDV. It's recommended to visit the website that describes in detail these procedures for several operating systems.

[1] Wikipedia, *Agile Software Development*, December 2015; see https://en.wikipedia.org/wiki/Agile_software_development

2 The Overall Architecture of a Data Virtualization Solution

As there are many roads that lead to Rome, there are many approaches to design and develop a *JBoss Data Virtualization* (JDV) environment. But whatever the approach, the recommended result is always a layered architecture consisting of minimally four layers of views; see Figure 1.

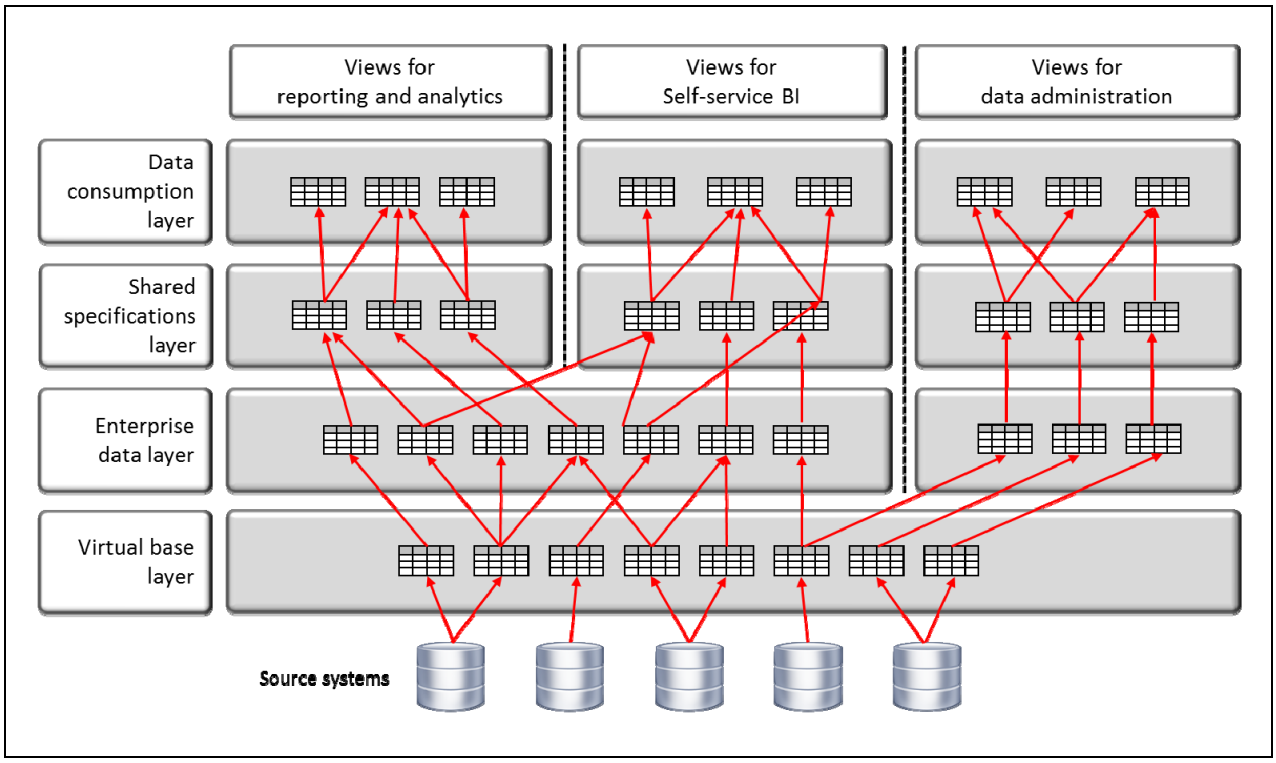

Figure 1 *A JBoss Data Virtualization environment consisting of four layers of views.*

The bottom layer of Figure 1 is linked to the *source systems*, such as SQL and NoSQL databases, SOAP/REST-based services, cloud applications, and files in JSON or XML formats. The bottom layer is also responsible for improving the data quality by applying cleansing rules. Applications, reports, and users, which are called *data consumers* in this whitepaper, retrieve data by accessing the top layer of views. All the layers are involved in transforming and restructuring the data to fit the needs of these data consumers.

In this architecture each view layer has its purpose (starting at the bottom):

- **Virtual Base Layer:** The virtual base layer contains views that contain the data stored in source systems. For each physical table or file in a source system a view is created. Each view definition may contain cleansing specifications to improve the quality of the data coming from the source systems. Besides correcting the data, the virtual contents of such a view is identical to the contents of the source system.

- **Enterprise Data Layer:** Views at the second layer present an integrated view of all the data in the source systems, hence the name *enterprise* data layer. The structure of each view is "neutral." In other words, it's not aimed at the needs of one data consumer, but at supporting as many forms

of usage possible. If possible, each view is structured according to the third normal form.

- **Shared Specifications Layer:** To avoid too many duplicate and possibly inconsistent specifications in the data consumption views, the third layer contains *shared specifications*. The purpose of this layer is to make the environment as agile as possible by avoiding duplication of specifications. Optionally, the shared specifications layer contains authorization specifications (who is allowed to use which view).

- **Data Consumption Layer:** The structure of each view at the data consumption layer focuses on simplifying data access for data consumers. For example, for some data consumers it may be useful to see the data organized as a star schema, whereas others prefer to see all the data they require in one view consisting of a large set of columns. Filters, projections, transformations, and aggregations are specified at this data consumption layer show data consumers only relevant data at the right aggregation level and in the right form.

Figure 1 also indicates three different forms of usage of the views: traditional usage, self-service usage, and administrative usage:

- *Traditional usage* relates to repetitive usage of views by data consumers for which the views are designed, developed, tested, and managed by IT specialists. Each view for traditional usage is commonly used by several data consumers and the data quality of its virtual content is important.

- *Self-service usage* relates to views developed by users themselves and not by IT specialists. Especially in the business intelligence sector, self-service development of reports has become very popular. Well-known tools are Qlikview, Tableau, and Tibco Spotfire. By allowing these users to develop their own views, their self-service capabilities are extended.

- *Administrative usage* refers to views developed by and for IT specialists who are responsible for data administration aspects, such as data quality and usage. For example, an administrative view may show the records with incorrect data values and another may show the usage of views over time by data consumers.

3 Top-Down, Bottom-Up, or Inside-Out?

Several approaches exist to design and develop the four view layers: bottom-up, top-down, and inside-out approach; see Figure 2.

- **Bottom-up approach:** With the bottom-up approach the structure of the source systems is the starting point. Views are designed towards the needs of the data consumers: first the virtual base layer is developed, then the enterprise layer, next the shared specifications layer, and finally the data consumption layer. The challenge of this top-down approach is that it's not always easy to come lose from the structure of the source systems. The data structures of the source systems are commonly designed to support their own applications as effective and efficiently as possible, and, moreover, some were designed a long time ago and have been adapted many times and have become somewhat unstructured over time. The bottom-up approach can result in views at the top layers with structures that are heavily influenced by the data structures of the source systems

and not so much by the needs of the data consumers.

- **Top-down approach:** With the top-down approach, design starts with analyzing the information needs of the data consumers which leads to the development of a set of views on the data consumption layer. Next, views on the shared specifications layer, the enterprise data layer, and the virtual base layer are developed to make the views on the data consumption layer work. So, the layers are designed from the top layer to the bottom layer. This is a very practical approach and data consumers get access to the data quickly. The risk of this approach is that eventually the entire environment contains many duplicate or similar specifications. Sharing of specifications is hard to organize and on the long-term it reduces the potential agility level of the entire JDV environment.

- **Inside-out approach:** With the inside-out approach the enterprise data layer is the first layer designed. The structure of the views are defined as neutral as possible with the goal to support the largest possible set of data consumer needs. Next, the views on the virtual base layer are defined and linked to the physical tables using a 1:1 approach. The views on the enterprise data layer are mapped to views on the virtual base layer. Afterwards, views on the data consumption layer are defined. The structure of these views is purely aimed at the data consumer needs. The ones defined are studied and sharable specifications are removed and implemented in views in the shared specifications layer. Inside-out is the preferred approach, but in real-life projects developers are sometimes forced to apply one of the other approaches or a combination of approaches.

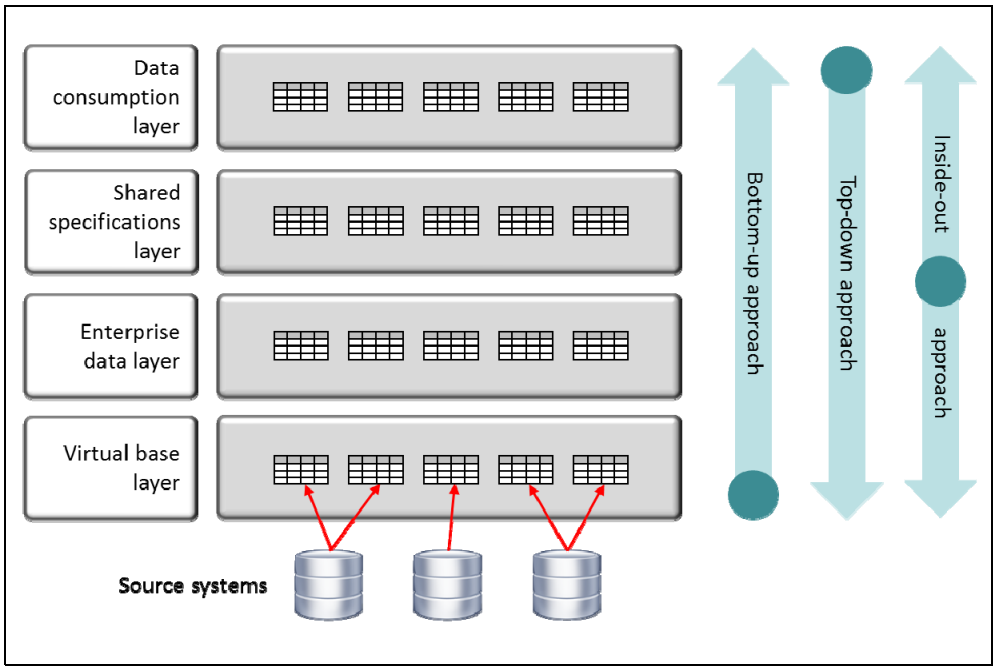

Figure 2 *Three different approaches to design and develop a JDV environment.*

4 Vertical or Horizontal?

Independently of the approach described in the previous section, another decision must be made on whether an entire layer is designed and developed before work starts on the next layer, or that a limited

number of views is defined in a layer before work on the next layer begins. The former approach is called the *horizontal approach* and the latter the *vertical approach*; see Figure 3. Using the terminology of *iterations*, with the vertical approach the iterations are short to very short, and with the horizontal approach the iterations can be long to very long.

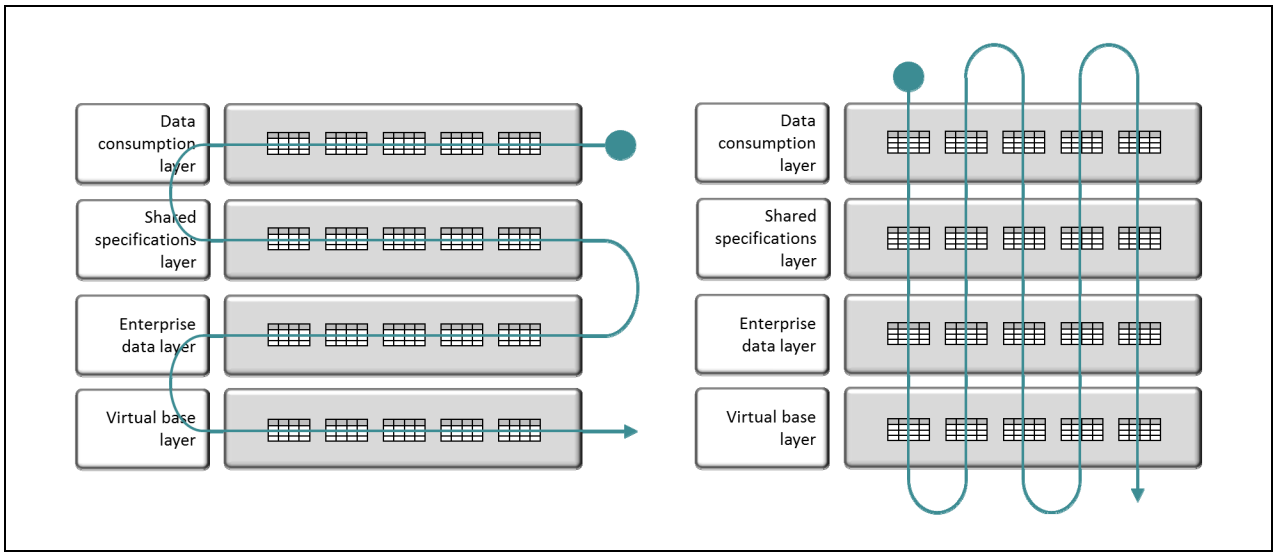

Figure 3 *The horizontal approach is shown on the left and the vertical approach on the right.*

With the horizontal approach a layer has to be completed before work starts on the next one. The horizontal approach corresponds with enterprise-wide approaches and waterfall[2] design techniques. The benefit of the horizontal approach is that it's easier to end up with a set of views that contains no or minimal redundant specifications.

With the vertical approach a limited number of views is defined within a layer before work starts on views in the next layer. The vertical approach fits very well with current *agile design techniques*[3] where development is highly iterative. Also, it works well for modern-day data warehouses where the requirement is to be able to develop more quickly and to change existing reports more quickly. The overall benefits of the vertical approach are: high productivity and agile solution. If users need data from some source system, the required views can be defined in a minimal amount of time. In other words, the time to react is short. In a green field situation, it will not take several months before the first reports can be developed when the vertical approach is used but rather days or weeks. In most data virtualization projects, the vertical approach is preferred because of its agility.

5 The Sample Database

In the next sections the step-by-step approach to design and develop a data virtualization environment is described. Many examples are used to explain the steps in detail. All of them use the same sample database. It's a production database for a fictitious online retail company called *World Class Movies*

[2] Wikipedia, *Waterfall Model*, December 2015; see https://en.wikipedia.org/wiki/Waterfall_model
[3] Wikipedia, *Agile Software Development*, December 2015; see https://en.wikipedia.org/wiki/Agile_software_development

(WCM) that offers movies for sale and rental. The database consists of a set of tables to keep track of customer information and information on sales and rentals.

This database is designed by Roland Bouman[4] and Jos van Dongen for their book *Pentaho Solutions; Business Intelligence and Data Warehousing with Pentaho and MySQL* and has also been used in the book *Data Virtualization for Business Intelligence Systems*[5]. Not all the tables in this database are used in this whitepaper, but only the ones displayed in the data model presented in Figure 4.

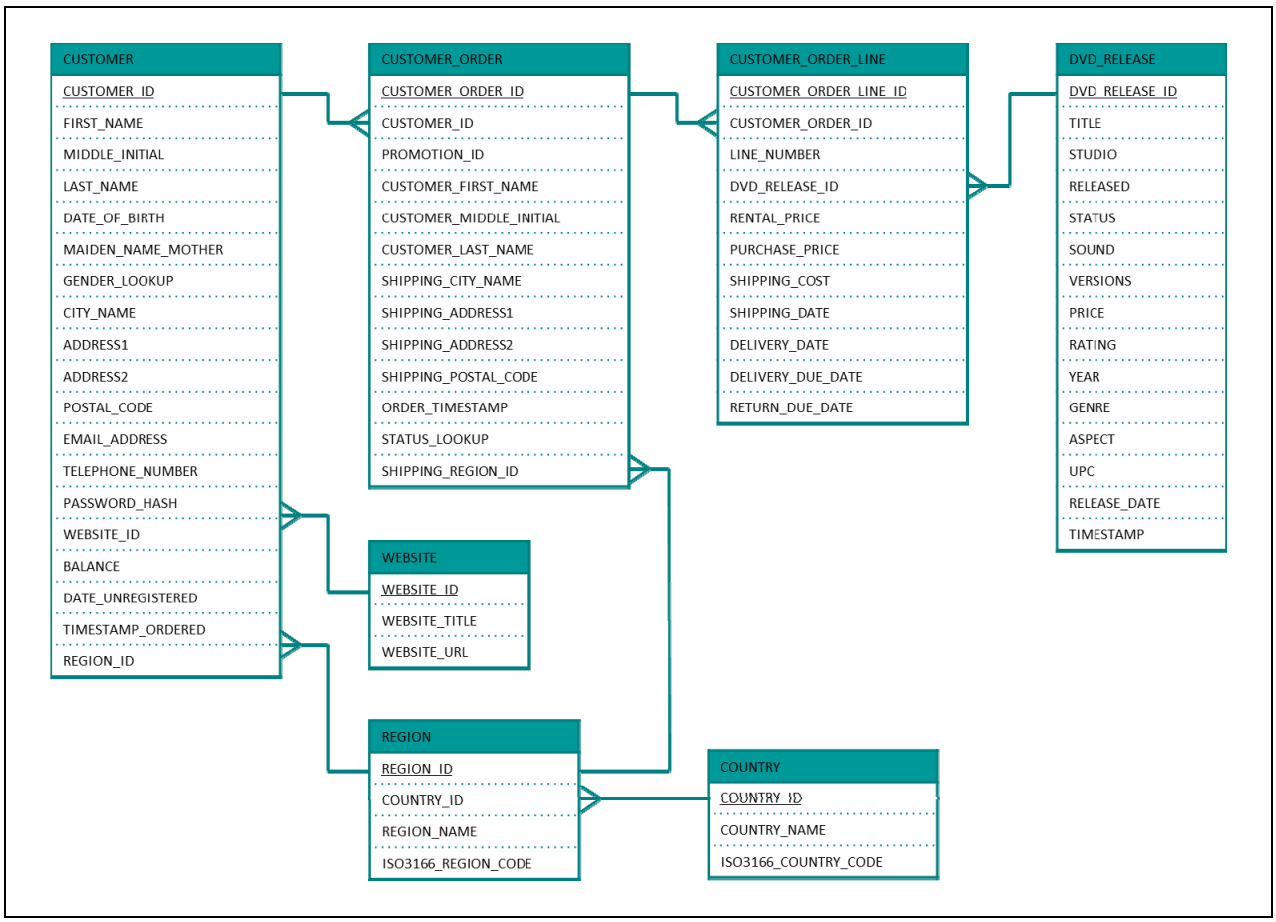

Figure 4 *The data model of the WCM sample database.*

6 Step 1: Designing and Implementing the Virtual Base Layer

Importing Source Systems – In this first layer, all the source systems required in the iteration are imported into JDV. Importing means that the technical specifications of a source system are imported. Potential source systems are SQL-based productions systems, data warehouses, and data marts, and also non-SQL based systems developed with Hadoop, NoSQL database servers, external files with data stored in JSON or XML format, and even spreadsheets. For most source systems this first step is relatively straightforward.

[4] R. Bouman and J. van Dongen, *Pentaho Solutions; Business Intelligence and Data Warehousing with Pentaho and MySQL*, John Wiley and Sons, Inc., 2009.

[5] R.F. van der Lans, *Data Virtualization for Business Intelligence Systems*, Morgan Kaufmann Publishers, 2012.

For the non-SQL ones it might be necessary to define how the non-relational data structures are mapped into flat relational views (commonly called *flattening* of the data).

The result of this importing process is that so-called *source tables* are developed. In JDV source tables are mere replicas of the source data. The content of the source table is identical to that of the underlying source system.

Cleansing Data with Data Virtualization is Different – Data stored in source systems is not always correct. Names are spelled incorrectly, numeric values are outside realistic boundaries, values in two fields have accidently been switched, stored values don't represent reality, and particular values or rows are completely missing. If no actions are taken, JDV will present this incorrect data to the data consumers. The consequence is that business decisions are made on incorrect data.

In more classic data warehouse environments ETL scripts are used to copy data from source systems to the data warehouse and data marts. Rules can be implemented within those ETL scripts to check incorrect data. The effect is that correct data is copied to the data warehouse and the incorrect data to a separate data store (the garbage bin). Afterwards, this garbage bin is studied to determine what to do with the incorrect data. Should more transformation rules be implemented or should the owner of the data be warned about the incorrectness of the data and asked to correct it?

This style of handling incorrect data doesn't work for data virtualization servers. When on-demand transformation is used, each mapping has only one result and this result is returned to the data consumer. In other words, there is no 'second' data store in which incorrect data can be found after executing the transformations. Therefore, data cleansing is executed differently in JDV.

To cleanse the data, *rules* must be specified. Because no filters, transformations, and projections can be specified in source tables, a layer of views has to be defined on top of the source tables; one view for each source table (see Figure 5). These views include the rules to cleanse the data from the source systems before it's passed on to the next layer. So, technically, the virtual base layer consists of two layers of tables. The first layer consists of the source tables that extract data from the source systems available and the second layer contains for each source table a view with cleansing rules.

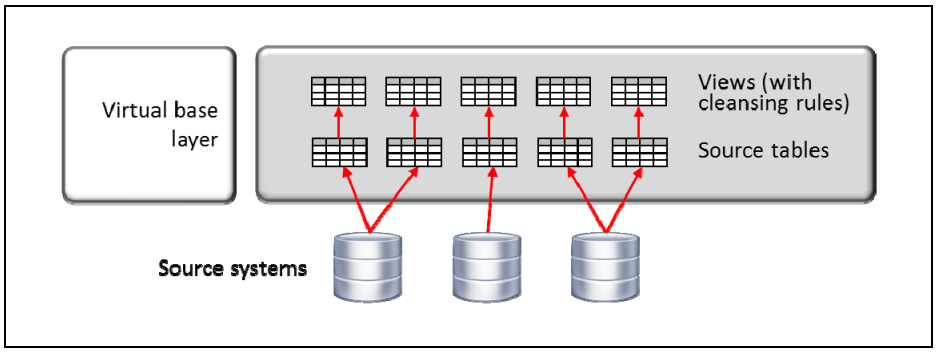

Figure 5 *Source tables are defined on the source systems, and for each source table a view is defined that contains data cleansing specifications.*

Note: The assumption is made here that all the data cleansing is handled by the views defined in the virtual base layer. It's however recommended to do as much of the data cleansing inside the source systems themselves and as close to the source as possible.

Five Approaches to Cleans Data – Five different approaches exist to handle incorrect data and to define cleansing rules in views:

- **Nothing:** No logic is added to detect incorrect data and data is returned to the data consumers uncorrected. In this case, it's the responsibility of the data consumers to detect incorrect data and to determine what to do with it. From the perspective of JDV, this is the easiest approach, but with regard to the quality of decision making, it isn't the preferred approach.

- **Filtering of rows:** With this approach cleansing rules are added to the view definitions in the virtual base layer with the intention to filter out incorrect data. Every row that contains one or more incorrect values is removed from the view's content. The result is that data consumers only see rows with correct data.

- **Filtering of values:** A slightly different version of filtering involves filtering values and not rows. If an incorrect value in a row is detected, the row remains in the view's content, but the value is removed. The value can be replaced by a null value or by a special code indicating that the original value is incorrect and has been replaced.

- **Flagging:** With flagging an extra column is added to the view (not available in the source table) that contains code indicating incorrect values. These codes in this extra column are called *flags*. For example, such a column can have the value `NAM,CIT` indicating that the values in the NAME and CITY columns are incorrect. A separate flag column can also be added for each column that can hold incorrect data. It's up to the data consumers to determine how to react to flags and the data.

- **Restoring:** With all the previous approaches incorrect data is not corrected. Replacing an incorrect value with a correct one, is called restoring. Restoring logic can be added to the definitions of the views in the virtual base layer.

The rest of this section contains examples of view definitions for the last four approaches.

Cleansing Data by Filtering Data – As indicated, two styles of filtering exist. One that filters out rows and one that filters out individual values. Usually, filtering rows is the easiest to implement. Cleansing rules are implemented as filters in the view definitions.

Example 1: The following rule applies to the `CUSTOMER_ORDER` table of the sample database: date values in the `ORDER_TIMESTAMP` column must all be greater than 31 December 1999, because the company didn't exist before that date. Here is the definition of the view on the `CUSTOMER_ORDER` table in which this rule is implemented as a filter:

```
CREATE   VIEW  VT1_CUSTOMER_ORDER_FILTERED_V1 AS
SELECT   *
FROM     CUSTOMER_ORDER
WHERE    ORDER_TIMESTAMP > DATE('1999-12-31')
```

Explanation: All the rows that are too old are removed from the content of the view. So, data consumers never see these rows.

Multiple rules can be implemented as filters in a view definition, as is shown in the next example.

Example 2: The following rules hold for the DVD_RELEASE table: the value X for RATING is incorrect, the column ASPECT must not contain the values LBX and VAR, and the column STATUS must be equal to one of the values Cancelled, Discontinued, Out, Pending, Postponed, and Recalled.

```
CREATE   VIEW  VT1_DVD_RELEASE_CHECKED_V1 AS
SELECT   *
FROM     DVD_RELEASE
WHERE    RATING <> 'X'
AND      ASPECT NOT IN ('LBX', 'VAR')
AND      STATUS IN ('Cancelled', 'Discontinued', 'Out', 'Pending', 'Postponed', 'Recalled')
```

The following table shows some of the rows and columns of the CUSTOMER table that are not part of the contents of the VT1_DVD_RELEASE_CHECKED view:

```
DVD_RELEASE_ID  RATING  ASPECT  STATUS
--------------  ------  ------  ------------
          2453  X       1.66:1  Discontinued
          2520  X       1.33:1  Out
         16769  NR      VAR     Out
         16770  NR      VAR     Out
        145377  UR      1.33:1  Discontineud
```

An alternative to filtering of rows is filtering of individual values. In this case, all the rows remain in the result, but incorrect values are removed and replaced by special values.

Example 3: Implement the rule specified in Example 1 as a filter on the ORDER_TIMESTAMP column.

```
CREATE   VIEW  VT1_CUSTOMER_ORDER_FILTERED_V2 AS
SELECT   CUSTOMER_ORDER_ID, CUSTOMER_ID, PROMOTION_ID, CUSTOMER_FIRST_NAME,
         CUSTOMER_MIDDLE_INITIAL, CUSTOMER_LAST_NAME, SHIPPING_CITY_NAME,
         SHIPPING_ADDRESS1, SHIPPING_ADDRESS2, SHIPPING_POSTAL_CODE,
         CASE
            WHEN ORDER_TIMESTAMP > DATE('1999-12-31') THEN ORDER_TIMESTAMP
            ELSE NULL END AS ORDER_TIMESTAMP_FILTERED,
         STATUS_LOOKUP, SHIPPING_REGION_ID
FROM     CUSTOMER_ORDER
```

Explanation: If the value for ORDER_TIMESTAMP is too old, it's replaced by a null value. Another value can be selected as well. The content of this view contains all the rows of the underlying table.

Example 4: Implement the three rules specified in Example 2 as filters for values.

```
CREATE   VIEW  VT1_DVD_RELEASE_CHECKED_V2 AS
SELECT   DVD_RELEASE_ID, TITLE, STUDIO, RELEASED,
         CASE
            WHEN STATUS IN ('Cancelled', 'Discontinued', 'Out',
                            'Pending', 'Postponed', 'Recalled') THEN STATUS
            ELSE NULL END AS STATUS_FILTERED,
         SOUND, VERSIONS, PRICE,
         CASE
            WHEN RATING <> 'X' THEN RATING
            ELSE NULL END AS RATING_FILTERED,
```

```
        YEAR, GENRE,
        CASE
           WHEN ASPECT NOT IN ('LBX', 'VAR') THEN ASPECT
           ELSE NULL END AS ASPECT_FILTERED,
        UPC, RELEASE_DATE, TIMESTAMP
FROM    DVD_RELEASE
```

This table shows a few rows and columns of the contents of this view:

```
DVD_RELEASE_ID  STATUS_FILTERED  RATING_FILTERED  ASPECT_FILTERED
--------------  ---------------  ---------------  ---------------
          2453  Discontinued     ?                1.66:1
          2520  Out              ?                1.33:1
         16769  Out              NR               ?
         16770  Out              NR               ?
        145377  ?                UR               1.33:1
```

Explanation: Incorrect values in the columns STATUS, RATING, and ASPECT are removed and null values are inserted.

Cleansing Data by Flagging the Data – Flagging of incorrect data in separate columns is very much like filtering of values. The rules can be implemented in a similar way.

Example 5: Define a view that implements the rule of Example 1 but now by using flagging.

```
CREATE   VIEW  VT1_CUSTOMER_ORDER_FLAGGED AS
SELECT   CUSTOMER_ORDER_ID, CUSTOMER_ID, PROMOTION_ID, CUSTOMER_FIRST_NAME,
         CUSTOMER_MIDDLE_INITIAL, CUSTOMER_LAST_NAME, SHIPPING_CITY_NAME,
         SHIPPING_ADDRESS1, SHIPPING_ADDRESS2, SHIPPING_POSTAL_CODE, ORDER_TIMESTAMP,
         CASE
            WHEN ORDER_TIMESTAMP > DATE('1999-12-31') THEN NULL
            ELSE 'Incorrect' END AS ORDER_TIMESTAMP_FLAG,
         STATUS_LOOKUP, SHIPPING_REGION_ID
FROM     CUSTOMER_ORDER
```

Explanation: The virtual content of this view includes the column ORDER_TIMESTAMP, which contains untransformed values plus a new column called ORDER_TIMESTAMP_FLAG. The essential difference between filtering of values and flagging is that with the former the incorrect values are removed and with flagging the incorrect values are still visible, but a special value indicates that the value is incorrect. It's up to the consumer of the data to determine what to do with this data.

For each column for which a cleansing rule is specified, a separate flag column can be defined. The next example shows how flagging for multiple columns can be applied by using only one column.

Example 6: Implement the three rules specified in Example 2; use flagging in one extra column.

```
CREATE   VIEW  VT1_DVD_RELEASE_FLAGGED AS
SELECT   DVD_RELEASE_ID, TITLE, STUDIO, RELEASED, STATUS, SOUND, VERSIONS, PRICE,
         RATING, YEAR, GENRE, ASPECT, UPC, RELEASE_DATE, TIMESTAMP,
         CASE
            WHEN STATUS IN ('Cancelled', 'Discontinued', 'Out',
                            'Pending', 'Postponed', 'Recalled') THEN ''
            ELSE 'STA,' END ||
```

```
          CASE
             WHEN RATING <> 'X' THEN ''
             ELSE 'RAT,' END ||
          CASE
             WHEN ASPECT NOT IN ('LBX', 'VAR') THEN ''
             ELSE 'ASP,' END AS FLAGS
FROM      DVD_RELEASE
```

This table shows a few rows and columns of the contents of this view:

```
DVD_RELEASE_ID  STATUS        RATING  ASPECT  FLAGS
--------------  ------------  ------  ------  -----
          2453  Discontinued  X       1.66:1  RAT,
          2520  Out           X       1.33:1  RAT,
         16769  Out           NR      VAR     ASP,
         16770  Out           NR      VAR     ASP,
        145377  Discontineud  UR      1.33:1  STA,
```

Explanation: If a row contains an incorrect status, rating, and aspect, the value is set to `STA,RAT,ASP,` in the `FLAGS` column.

Cleansing Data by Restoring the Data – When filtering or flagging is used, data is not corrected. Incorrect data is removed or flagged as being incorrect. Views can be used to replace incorrect data by correct data. This technique can only be used for restoring misspelled data.

Example 7: Define a view that implements rules for the columns `STATUS`, `SOUND`, `RATING`, and `ASPECT`. Restore as many values in those columns as possible.

```
CREATE   VIEW  VT1_DVD_RELEASE_RESTORED AS
SELECT   DVD_RELEASE_ID, TITLE, STUDIO, RELEASED,
         CASE STATUS
            WHEN 'Discontineud' THEN 'Discontinued'
            ELSE STATUS END AS STATUS_EDITED,
         CASE SOUND
            WHEN '4.0 DTS/4.0' THEN '4.0/DTS'
            WHEN '4.0/4.0 DTS' THEN '4.0/DTS'
            WHEN '4.1 DTS/4.1' THEN '4.1/DTS'
            WHEN '4.1/DTS 4.1' THEN '4.1/DTS'
            WHEN 'DTS 5.0'     THEN '5.0/DTS'
            WHEN '5.0/DTS 5.0' THEN '5.0/DTS'
            WHEN '5.1 DTS'     THEN '5.1/DTS'
            WHEN '5.1 HD-DTS'  THEN '5.1 DTS-HD'
            ELSE SOUND END AS SOUND_EDITED,
         VERSIONS, PRICE,
         CASE RATING
            WHEN 'PA'   THEN 'PG'
            WHEN 'UN'   THEN 'UR'
            WHEN 'UR/R' THEN 'R/UR'
            ELSE RATING END AS RATING_EDITED,
         YEAR, GENRE,
         CASE ASPECT
            WHEN '1.781' THEN '1.78:1'
            ELSE ASPECT END AS ASPECT_EDITED,
         UPC, RELEASE_DATE, TIMESTAMP
FROM     DVD_RELEASE
```

This table shows a few rows and columns of the contents of this view:

```
DVD_RELEASE_ID  STATUS_EDITED  SOUND_EDITED  RATING_EDITED  ASPECT_EDITED
--------------  -------------  ------------  -------------  -------------
         18739  Out            5.0/DTS       NR             1.85:1
         67143  Out            4.0/DTS       NR             1.78:1
        117175  Recalled       5.1           R/UR           1.78:1
        127720  Out            5.1           R              1.78:1
        144253  Out            2.0           PG             1.33:1
        145377  Discontinued   2.0           UR             1.33:1
```

Explanation: The column `STATUS` contains one misspelled value `Discontineud`, which is transformed to `Discontinued`. For the columns `SOUND`, `ASPECT`, and `RATING` the same type of transformation is applied. This solution works well, but its restriction is that new incorrect values entered in the source system are not automatically discovered and added to the view definition. Someone has to keep an eye on this.

Another approach is to create a dedicated table that holds all the incorrect values and their correct counterparts. For example, the following `SOUND_CODES` table with the following content can be developed:

```
SOUND_MISSPELLED  SOUND_CORRECT
----------------  -------------
4.0 DTS/4.0       4.0/DTS
4.0/4.0 DTS       4.0/DTS
4.1 DTS/4.1       4.1/DTS
4.1/DTS 4.1       4.1/DTS
DTS 5.0           5.0/DTS
5.0/DTS 5.0       5.0/DTS
5.1 DTS           5.1/DTS
5.1 HD-DTS        5.1 DTS-HD
```

Instead of the long `CASE` expression, the transformation looks like this:

```
IFNULL(SELECT   SOUND_CORRECT AS SOUND_EDITED
       FROM     SOUND_CODES
       WHERE    SOUND_INCORRECT = SOUND), SOUND)
```

Explanation: If the code is misspelled, the expression returns the correct code from the `SOUND_CODES` table, else it returns the value of the `SOUND` column itself.

The advantage of the approach with a translation table is that if new misspelled codes are detected, the view definition doesn't have to be changed, only the `SOUND_CODES` table has to be updated. It's an approach that is easier to maintain.

In the examples the number of misspelled values is relatively low. The problem is harder to fix when it concerns names such as those of companies, customers, and cities. Neither solution (case expression or translation table) works in this situation. To determine whether a name has been spelled incorrectly, more advanced technology, such as fuzzy matching, is needed. Access to data sets with correct data may be needed as well. For example, even if the names of a city, street, and state are spelled correctly, the three might still not belong to each other. The only way to detect that the address doesn't exist, is by accessing a list of all the correct addresses.

7 Step 2: Designing and Implementing the Enterprise Data Layer

The Neutral Tables – The structures of the views in the enterprise data layer represent all the data in the most neutral form possible. When data consumers access this layer they can't see that data is coming from multiple source systems, that the structure of the views in the virtual base layer is not perfect, or that some data is not even stored in non-relational data structures. The enterprise data layer hides all this. The views in this layer are sometimes called the *canonical data model*.

It's recommended to design these views as much as possible according to the *third normal form*. In such a table "every non-key attribute must be dependent on the key, the whole key, and nothing but the key." This is a slight adaptation of William Kent's[6] definition of the third normal form. The effect is that the virtual content of all the views together is without any redundancy. This neutral structure is probably not perfect for every data consumer accessing these views, but it's adequate for most forms of usage.

Note: Views in the enterprise data layer must be defined on views in the virtual base layer. Similar rules apply for the other layers. However, technically it's impossible to enforce this mechanism. JDV can't stop developers to define views in the data consumption layer directly linked to the source tables. Therefore, this must be verified periodically.

Integration of Source Systems – Data that logically belongs together, but is stored in separate source systems, must be integrated. The views in the enterprise data layer are responsible for all the data integration. For example, if multiple source systems contain customer data, then one view in this layer integrates this data.

Integration can be based on join or union operations. *Join integration* means that two source systems contain comparable sets of objects but have different sets of attributes. For example, one source system contains address related data on customers and the other contains consumption data on customers. A join in the definition of the customer view can be used to integrate them.

Example 8: Imagine that the WCM company operates two systems that contain customer data. One system contains all the name and address related data and the other one the other columns. In this case, a *full outer join* is required to make them look like one table:

```
CREATE   VIEW  VT2_CUSTOMER_WIDE AS
SELECT   CN.FIRST_NAME, CN.MIDDLE_INITIAL, CN.LAST_NAME, CN.DATE_OF_BIRTH,
         CN.MAIDEN_NAME_MOTHER, CN.GENDER_LOOKUP, CN.CITY_NAME, CN.ADDRESS1, CN.ADDRESS2,
         CN.POSTAL_CODE, CN.EMAIL_ADDRESS, CN.TELEPHONE_NUMBER, CN.PASSWORD_HASH,
         CN.WEBSITE_ID, CO.BALANCE, CO.DATE_REGISTERED, CO.DATE_UNREGISTERED
FROM     CUSTOMER_NAME AS CN
         LEFT OUTER JOIN CUSTOMER_OTHER AS CO ON CN.CUSTOMER_ID = CO.CUSTOMER_ID
```

In contrary, *union integration* is required when two source systems contain exclusive sets of objects and comparable attributes. With a union operator in the view definition these two source systems can be integrated.

Example 9: Imagine that the WCM company owns two systems for storing customer data. One contains all the customers from the northern region and the other all the customers from the southern region. A

[6] W. Kent, *A Simple Guide to Five Normal Forms in Relational Database Theory*, Communications of the ACM, Volume 26, 1983.

union operator is required to make them look like one table:

```
CREATE   VIEW  VT2_CUSTOMER_ALL AS
SELECT   *
FROM     CUSTOMER_NORTHERN
UNION ALL
SELECT   *
FROM     CUSTOMER_SOUTHERN
```

When the two tables do not really have the same set of columns, a missing columns in one table must be filled with null values or some other meaningful values.

The Need for Master Data – Some join integrations cannot be handled by simply including a join operation in the view definition. For example, imagine that the WCM organization has two IT systems in which customer data is stored, but the customer IDs don't match. In this case, *master data* is indispensable. If the number of customers is limited, it's sufficient to develop a table that holds the IDs of the two customer systems and that indicates how the customers relate to each other. This table can be stored in one of the existing source systems and requires a source table and a view. Next, it can be included to join WCM's two customer tables.

When several master data solutions are required and when the amount of master data is significant, a more structural solution is required. In other words, a *master data management system* is needed that contains sufficient data to relate the right customer from one source system to the right customer in the other source system. Such an MDM system becomes a source system for the virtual base layer. Technically, if this MDM allows the master data to be accessed through a SQL interface, then this is preferred over a more service oriented interface, such as SOAP or REST, because a SQL interface is more efficient.

Combining Too Many Tables – A special situation arises when one view in the enterprise data layer points to a large number of views and source tables in the virtual base layer. For example, imagine that the WCM company delivers DVDs to hundred stores spread across the country. Imagine also that each store runs its own local database for storing sales data. If the rules of this section are followed blindly, then hundred views must be defined where each one points to a physical table at one of the stores. To present an overview of all the sales, the view in the enterprise data layer becomes a union of all these hundred source systems. This is an unpractical solution. A more efficient implementation, however, is to use the *multi-source model* feature of JDV. A definition would look like this:

```
<vdb name="vdbname" version="1">
    <model visible="true" type="PHYSICAL" name="Customers" path="/Test/Customers.xmi">
        <property name="multisource" value="true"/>
        <source name="Customers_NewYork"
            translator-name="oracle" connection-jndi-name="newyork-customers"/>
        <source name="Customers_Boston"
            translator-name="oracle" connection-jndi-name="boston-customers"/>
        <source name="Customers_Phoenix"
            translator-name="oracle" connection-jndi-name="phoenix-customers"/>
    </model>
</vdb>
```

This solution works when the tables in all the remote source systems have the same schema. Compared to the source systems, each view has one extra column that indicates for each record its location.

8 Step 3: Designing and Implementing the Shared Specifications Layer

Views defined in the shared specifications layer present the data visible in the enterprise data layer, but with data structures that are easier to use for the data consumers. Views in the shared specifications layer don't change the data nor do they filter out data. It's primarily a re-arrangement of the data.

In principle, the shared specifications layer is not required. Technically, the layer on top of the shared specifications layer, the data consumption layer, can be developed straight on the enterprise data layer. It's recommended to introduce the shared specifications layer for reasons of performance, sharing specifications, and processing derived data.

Shared Specifications Layer for Performance Optimization – If data consumers prefer to see the data in a star schema or snowflake pattern, which many do, defining views directly on the normalized views of the enterprise data layer can be very complex. The result is that the underlying source system has to process complex SQL statements that may be slow. Therefore, it's recommended to define a very generic form of a star schema structure in the shared specifications layer. When a view is defined, it must be cached to speed up performance. The refresh rate of a cache is determined by the required data latency. These star schema views must contain all the data and not only the current data. The reason is that the more data they contain, the more data consumers and the more users can make use of the same set of views and, therefore, the same caches. More on caches in Section 12.

Example 10: Define a view in the shared specifications layer that contains *fact data* from the sample database. The assumption is made that views exist on the enterprise data layer with the same data structure as the tables specified in Figure 4.

```
CREATE   VIEW  VT3_TRANSACTIONS AS
SELECT   CO.CUSTOMER_ID, COL.DVD_RELEASE_ID, CO.ORDER_TIMESTAMP AS DATE_ID,
         COL.RENTAL_PRICE, COL.PURCHASE_PRICE, COL.SHIPPING_COST
FROM     VT2_CUSTOMER_ORDER_LINE AS COL LEFT OUTER JOIN VT2_CUSTOMER_ORDER AS CO
         ON CO.CUSTOMER_ORDER_ID = COL.CUSTOMER_ORDER_ID
```

Example 11: Define a view in the shared specifications layer that contains *dimension data* from the sample database. The assumption is made that views exist on the enterprise data layer with the same data structure as the tables specified in Figure 4.

```
CREATE   VIEW  VT3_CUSTOMERS AS
SELECT   C.FIRST_NAME, C.MIDDLE_INITIAL, C.LAST_NAME, C.DATE_OF_BIRTH,
         C.MAIDEN_NAME_MOTHER, C.GENDER_LOOKUP, C.CITY_NAME, C.ADDRESS1, C.ADDRESS2,
         C.POSTAL_CODE, C.EMAIL_ADDRESS, C.TELEPHONE_NUMBER, C.PASSWORD_HASH,
         W.WEBSITE_ID, W.WEBSITE_TITLE, W.WEBSITE_URL,
         C.BALANCE, C.DATE_REGISTERED, C.DATE_UNREGISTERED, C.TIMESTAMP_CHANGED,
         R.REGION_ID, R.REGION_NAME, R.ISO3166_REGION_CODE,
         CN.COUNTRY_NAME, CN.ISO3166_COUNTRY_CODE
FROM     VT2_CUSTOMER AS C
         LEFT OUTER JOIN VT2_WEBSITE AS W ON C.WEBSITE_ID = W.WEBSITE_ID
         LEFT OUTER JOIN VT2_REGION AS R ON C.REGION_ID = R.REGION_ID
         LEFT OUTER JOIN VT2_COUNTRY AS CN ON R.COUNTRY_ID = CN.COUNTRY_ID
```

Both queries inside the view definitions are not simple. it becomes really complex when a data consumer joins these two views with an aggregation.

As can be seen in the definition of the virtual fact table, a dimension is required that holds date/time-related data. The recommendation is to develop such a table and to store it in the same database in which all the other cached tables are stored. This avoids a federated join.

Shared Specifications Layer for Sharing Specifications – Many views defined in the data consumption layer may share identical specifications. For example, imagine that two data consumers are looking at different columns of the same view of the enterprise data layer. One sees the data in an aggregated form and the other doesn't. Still, both may have identical filters in their view definitions, such as show only rows that have been added last year and only for customers from the northern region. It's technically not incorrect to have both filters in both view definitions, but it's more efficient to include these common specifications in a view on the shared specifications layer. This leads to sharing of specifications, and sharing eases maintenance. If the rule that indicates whether a customer is based in the northern or southern region changes, it has to be changed in one view only.

Shared Specifications Layer for Derived Data – Views in the shared specifications layer may include columns containing derived data. For example, when birth dates are available in a view of the enterprise data layer and the data consumers want to see age as well, an extra column can be defined in the view that shows that age. Or, when a view shows temperatures in Fahrenheit, the view in the shared specifications layer contains the temperature in Celsius as well.

Example 12: Define a new view in the shared specifications layer that is defined on the `VT2_CUSTOMER` view and contains a derived column, namely the date of the Sunday following the `TIMESTAMP_CHANGED` column.

```
CREATE   VIEW  VT3_CUSTOMER AS
SELECT   CUSTOMER_ID, FIRST_NAME, MIDDLE_INITIAL, LAST_NAME, DATE_OF_BIRTH,
         MAIDEN_NAME_MOTHER, GENDER_LOOKUP, CITY_NAME, ADDRESS1, ADDRESS2,
         POSTAL_CODE, EMAIL_ADDRESS, TELEPHONE_NUMBER, PASSWORD_HASH, WEBSITE_ID,
         BALANCE, DATE_REGISTERED, DATE_UNREGISTERED, TIMESTAMP_CHANGED, REGION_ID,
         CASE WHEN TIMESTAMP_CHANGED <> '9999-12-31 00:00:00'
            THEN CASE DAYOFWEEK(TIMESTAMP_CHANGED)
               WHEN 1 THEN TIMESTAMP_CHANGED + INTERVAL '7' DAY
               WHEN 2 THEN TIMESTAMP_CHANGED + INTERVAL '6' DAY
               WHEN 3 THEN TIMESTAMP_CHANGED + INTERVAL '5' DAY
               WHEN 4 THEN TIMESTAMP_CHANGED + INTERVAL '4' DAY
               WHEN 5 THEN TIMESTAMP_CHANGED + INTERVAL '3' DAY
               WHEN 6 THEN TIMESTAMP_CHANGED + INTERVAL '2' DAY
               WHEN 7 THEN TIMESTAMP_CHANGED + INTERVAL '1' DAY
            END
         ELSE TIMESTAMP_CHANGED END AS NEXT_SUNDAY
FROM     VT2_CUSTOMER
```

The benefit of adding derived data is that all the data consumers interested in the `NEXT_SUNDAY` column use the same definition and see consistent results. In addition, data consumers don't have to reinvent the formula to calculate the date of the next Sunday. This is good for productivity and maintenance.

Note: Include derived columns for which the formula requires only access to values stored in the same row. Don't include aggregated data, move it to the data consumption layer.

9 Step 4: Designing and Implementing the Data Consumption Layer

Defining the Views – In a way, this is the easiest step. Based on the needs of the data consumers the appropriate views are defined. If data consumers require a star schema, then a set of views is defined that form a star schema; if they prefer a snowflake-like organization of the data, then that's what they will get; and if data consumers prefer one big and wide view that contains all the data they need, then such a view is developed. This approach leads to several views in the data consumption layer that overlap content-wise, but that's no problem because there's no storage involved.

Try to identify common specifications in the views in this layer. If they exist, move them to the underlying shared specifications layer.

Defining Primary and Foreign Keys – Many tools for reporting and analytics need to know the primary and foreign keys of the tables they're accessing. Therefore, define on the views in the data consumption layer the proper primary and foreign keys. The key definitions help the tools to generate more optimized code for accessing the views.

Publishing Views – Before a view can be accessed, it has to be *published*. Publishing means that the content of the view becomes available for use by any data consumer. Publication involves defining the technical APIs and languages through which the view must be accessible. Technical interfaces are for instance JDBC/SQL, SOAP, and REST. Which API must be selected, depends on the tools and applications accessing the view. Usually, they require a specific API.

Data Security – When views are published, security aspects can be defined. Determine in cooperation with the local security experts within the organization which security mechanisms must be deployed.

10 Step 5: Designing and Implementing Views for Self-Service Usage

What is Business-Driven BI? – Many different forms of BI exist. All these forms can be divided in two categories: *IT-driven BI* and *Business-driven BI*. With IT-driven BI the reports are designed and developed (partially or fully) by IT and must be tested, governed, and audited. Typically, these reports have to be reproducible for years to come. Some of them must be distributed to regulators, customers, or suppliers. Typical terms associated with IT-driven BI are: industrial scale, specification-driven process, professionally tested, SLA metrics, governance, and control.

Business-driven BI relates to all the reports designed and developed by business users. Self-service BI, investigative analytics, data discovery, and data science are all forms of business-driven BI. Typical characteristics of this second form of BI are: self-service, agile, non-governed, non-auditable, investigative, one-time usage, no or minimal sharing.

The views defined in Steps 1 to 4 support IT-driven BI. For business-driven forms of BI, it's important that an area within the data consumption and shared specifications layers is reserved for self-service development of views. Here users can define their own views on top of views belonging to the enterprise data layer. It's not recommended to allow them to define views that access source systems directly, because they would have to handle incorrect data and integrate data from source systems. This can be

highly complex and will cost them valuable time. In addition, it can lead to incorrectly defined views which leads to incorrect report results.

Monitoring and Management – It's important that BI specialists verify and monitor the definitions of the user-defined views and their usage. Things to check are:

- Do users use the most efficient code to define the views?
- Do users join views in the enterprise data layer correctly?
- Do users reinvent the wheel over and over again?
- Do users really use the right views?

As SQL specialists know, queries can be written in many different ways. Some formulations are efficient and others not. Users don't always know the most efficient formulation. This can lead to unnecessarily slow queries. By monitoring the query performance, specialists can determine that some views are always slow. They can then study the definitions and see if they can come up with a faster solution. If so, more efficient formulations can be proposed to the users. The specialists can also assist users with implementing more efficient view definitions.

Due to this monitoring, development of these views for self-service usage becomes a very cooperative process in which users and BI specialists work together closely.

Reactive Development – Note that all the work by BI specialists on this self-service environment can be called *reactive*. Views in this environment are not first defined and then tested, tuned, and optimized before they can be used. On the contrary, after users define their own views they can use them right away. It's only afterwards that the specialists react. Monitoring is used to determine whether a reaction by the specialists is required. This is very different from development of the views described in Sections 8 and 9. These views become accessible after they have been tested and optimized.

11 Step 6: Designing and Implementing Views to Manage Data Quality

As indicated in Section 6, working with data virtualization technology such as JDV is different from working with ETL technology. Especially handling of data quality issues is different. Samples 1 to 7 describe how views can be defined that filter out or restore incorrect data. In these views incorrect data is not visible, except in the views that implement flagging.

But what if data quality specialists want to see false, missing, and misspelled data values, because if they can, they can go to the owners of the source systems to discuss how the incorrect data can be fixed. The result of such an exercise is that the overall quality of data improves.

To support data quality specialists, dedicated views can be defined that show all the incorrect data. These views show the filtered, flagged, or incorrect data. Defining this type of view is relatively easy as the next example illustrates.

Example 13: Define a view that shows all the data filtered out in Example 2.

```
CREATE   VIEW  VT2_INCORRECT_DVD_RELEASE AS
SELECT   *
FROM     DVD_RELEASE
EXCEPT
SELECT   *
FROM     VT1_DVD_RELEASE_CHECKED_V1
```

Explanation: The result of the first part of the query returns *all* the DVD releases. The second part returns all the DVD releases that are correct. If they are subtracted from each other, what remains are the rows that contain incorrect data. The same approach works for filtering and restoring of values.

12 Step 7: Optimizing Performance

Several techniques exist to improve the query performance of the four layers of views. This section discusses a few query optimization techniques. For more techniques and for more detailed descriptions we refer to the manuals of JDV and the database servers used by the source systems.

Caching Views – Caching views is one of the most powerful techniques offered by JDV to improve query performance. But improving query performance is not the only reason for defining caches. There are many practical reasons why views must be cached:

- **Slow Source Systems:** Some source systems can be very slow when being accessed. For example, some cloud applications have a one-record-at-a-time interface which is commonly quite slow. It's better to cache the view that accesses such an application. This cache is refreshed with a frequency that fits the needs of all users accessing this view.

- **Network Delays:** Accessing a remote source system can be exceedingly slow due to insufficient bandwidth or an unreliable network. Both can be reasons for caching the views pointing to these source systems.

- **Low Availability:** Data consumers may need access to the data from a source system 24 hours a day. If, however, that source system shuts down every evening at 8pm and is started again the next day at 6am, a cache on the view can raise the data's availability to the required 24 hours.

- **Complex View Definitions:** Most views in the virtual base layer are rather straightforward, but not all of them. The complexity of the cleansing rules can lead to poor query performance, or the source system is not relational but hierarchical and the processing required to flatten the data is complex and time-consuming. In both situations, a cache is recommended.

Note that some data consumers need to work with real-time data and some need to insert, update, and delete data in a source system. In both cases a cache cannot be defined. If there are data consumers with these requirements and others that need fast query access, two views can be defined on the same source system. Both will have the same data structure, but one isn't cached (for the data consumers who want to see real-time data) and one is (for the data consumers who want to run reports).

Guidelines for Caching Views – When caches are needed, define them on the views of the shared specifications layer. The benefit is that all the processing by the source systems and the cleansing of the data in the views (in the virtual base layer), the integration and normalization of data (in the enterprise data layer), and the restructuring of the data (in the shared specifications layer) have taken place.

Caches can also be defined on the views of the data consumption layer, but if it's possible, define them on the shared specifications layer, because then a larger group of data consumers benefit from a cache.

When particular views are used together often by data consumers, store their caches in the same database to avoid distributed join processing.

Defining Indexes on Cached Tables – When a cache is physically stored in a SQL database, JDV doesn't define indexes on it. It's recommended to do so. If a primary key exists on the cached table, define a unique index on it. Also define indexes on columns that are foreign keys pointing to other cached views. This helps the query optimizer to generate an optimal query plan. Moreover, define indexes on columns that are aggregated often and on columns on which many filters are applied. Note that adding extra indexes can slow down the refreshing of the caches somewhat.

Developing indexes on caches can't be done from within JDV. This must be done using the SQL database server itself. When extra indexes are defined, be sure that the right parameters are set; see the respective manuals of the SQL database servers.

Update Statistical Information – Query performance of a SQL database server can be poor when the *statistical information* on the tables and indexes is out of date, or in other words, if it doesn't reflect the current situation. Because the query optimizer of a SQL database server bases its query plan on this statistical information, it's crucial that it's always as up to date as possible. Out-of-date statistical information can lead to poor query performance. Therefore, it's important that the statistical information for cached views is up to date. After every refresh of a cache an update of the statistical information must be considered. See the respective manuals of the SQL database servers to determine how statistical information can be updated.

In addition, JDV itself needs this statistical information on source tables to optimize the queries it executes on the source systems. Therefore, it's important that statistical information is supplied on source tables to get optimal query plans.

Use Performance Improving Features – Each SQL database server offers a wide range of features to improve performance. Tables can be partitioned, buffers can be enlarged and tuned, table space parameters can be optimized, tables can be moved to faster storage technology, and so on. Check with database administrators to see if they can tune and optimize access to the cached tables.

Note: Whatever feature of the database server is used to optimize performance, never change the structure or the content of the cached views. This can damage the operation on the cached views.

Other Performance Improving Features – Network delay can influence performance negatively. Imagine that JDV runs on one server and a source system on another. Imagine also that this data source cannot execute any SQL operations. In other words, JDV has to retrieve all the data from the source system across the network and do all the query processing on its own server. To optimize network traffic, two instances of JDV may be installed; see Figure 6. The effect is that JDV running on the local server will send the entire

query to JDV running on the remote server. This instance extracts all the data from the source system, processes the query, and only returns the result of the query to the local service. This can dramatically reduce the amount of data to be transmitted across the network.

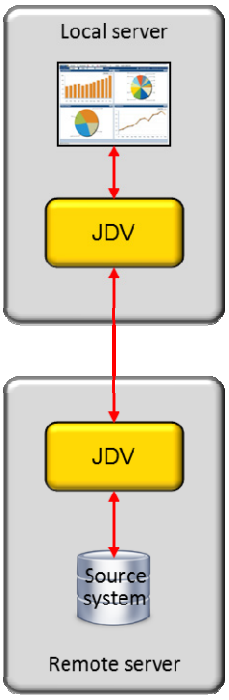

Figure 6 *To optimize network traffic, it may be necessary to install multiple instances of JDV.*

Final Remark – Many other techniques exist to improve performance. It would be impossible to describe them all in this whitepaper. But whatever technique is used, be sure that the technique doesn't minimize the agility level of the data virtualization environment.

13 Overview of Red Hat JBoss Data Virtualization

Red Hat offers the only open source alternatives to enterprises looking to adopt and implement data virtualization. The open source subscription model lowers the cost as adoption barrier, while open community based innovation offers open, non-proprietary option. Following key features and capabilities make JBoss Data Virtualization a worthy choice.

Model Driven Development – JBoss Data Virtualization includes Teiid Designer (see Figure 7), an Eclipse-based graphical tool, which models, analyzes, integrates, and tests multiple data sources to produce relational, XML, and web service views that show business data without programming. You can map from data sources to target formats using a visual tool, as well as resolve semantic differences, create virtual data structures at a physical or logical level, and use declarative interfaces that are compatible with and optimized for your applications.

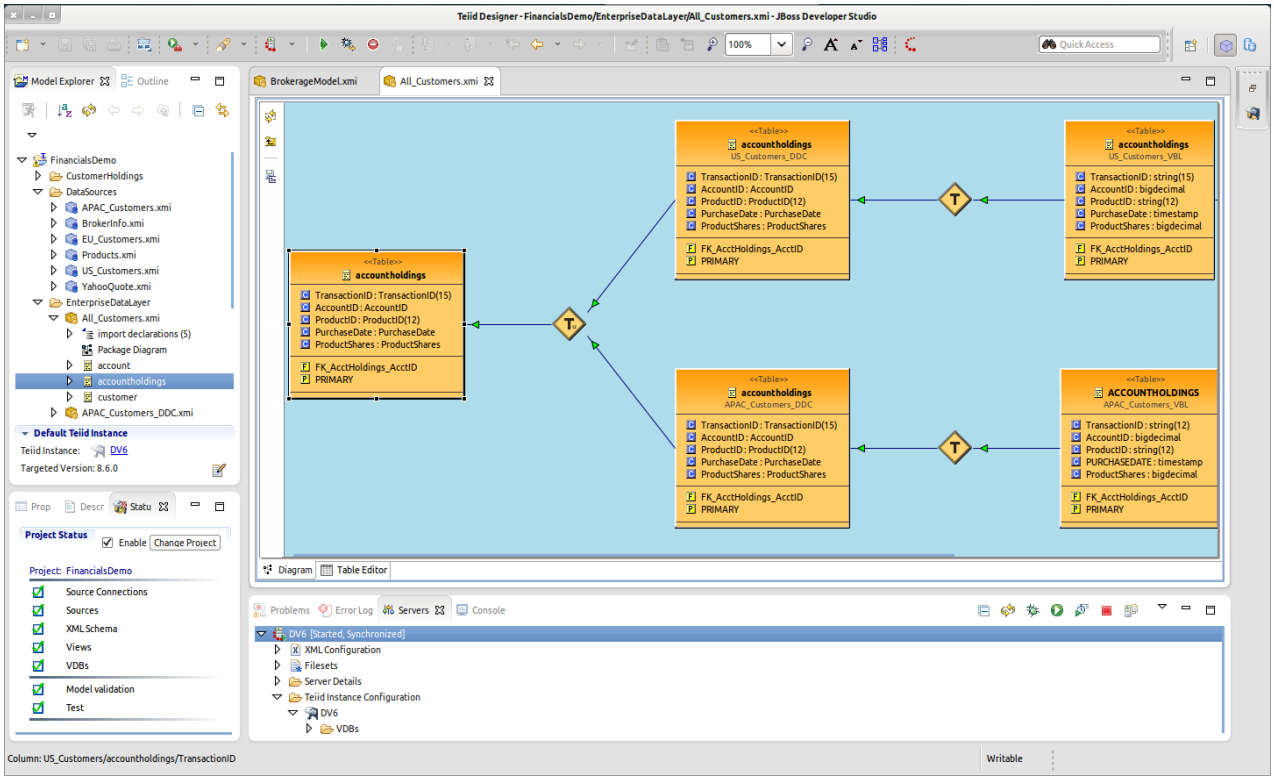

Figure 7 *JDV includes Teiid Designer, an Eclipse-based graphical tool.*

Centralized Data Security – JBoss Data Virtualization gives you the power to manage and monitor data services in a single unified environment and enforce and manage policies and roles across federated data for all data services. You can configure policies for data security, privacy, column-level data masking, and data sanitization of sensitive fields based on user roles. In addition to using the security capabilities of Red Hat JBoss Enterprise Application Platform, the software supports web services security and fine-grained access control for SQL data service and virtual table. In addition to column-level security, you get row-level security at the virtualization layer, which is independent of data sources. Transport and password encryption are available through SSL support.

Performance Optimization – JDV offers support for several advancing caching modes, including materialized views, result set caching, and code table caching that provide dramatic performance improvement. Configurable time-to-live, memory preferences, and updatability options are available for data caching.

Intelligent and automated techniques (e.g., cost- and rule-based query optimizer using information from source introspection, query capabilities, and constraints) include pushdown queries, dependent joins, projection minimization, partitioned aware unions, and copy criteria to optimize data sources join processing, and support for high-performance sub-queries. Comprehensive query trace is supported with manual plan override of automatic strategy selections for optimized query.

About the Author Rick F. van der Lans

Rick F. van der Lans is an independent analyst, consultant, author, and lecturer specializing in data warehousing, business intelligence, big data, database technology, and data virtualization. He works for R20/Consultancy (www.r20.nl), a consultancy company he founded in 1987.

Rick is chairman of the annual European Enterprise Data and Business Intelligence Conference (organized annually in London). He writes for Techtarget.com[7], B-eye-Network.com[8] and other websites. In 2009 he introduced the business intelligence architecture called the *Data Delivery Platform*, which is an architecture based on data virtualization, in a number of articles[9] all published at B-eye-Network.com.

He has written several books. His latest book[10] *Data Virtualization for Business Intelligence Systems* was published in 2012. Published in 1987, his popular *Introduction to SQL*[11] was the first English book on the market devoted entirely to SQL. After more than twenty five years, this book is still being sold, and has been translated in several languages, including Chinese, German, Italian, and Dutch.

For more information please visit www.r20.nl, or email to rick@r20.nl. You can also get in touch with him via LinkedIn and via Twitter @Rick_vanderlans.

About Red Hat, Inc.

Red Hat is the world's leading provider of open source solutions, using a community-powered approach to provide reliable and high-performing cloud, virtualization, storage, Linux, and middleware technologies. Red Hat also offers award-winning support, training, and consulting services. Red Hat is an S&P company with more than 70 offices spanning the globe, empowering its customers' businesses.

[7] See http://www.techtarget.com/contributor/Rick-Van-Der-Lans
[8] See http://www.b-eye-network.com/channels/5087/articles/
[9] See http://www.b-eye-network.com/channels/5087/view/12495
[10] R.F. van der Lans, *Data Virtualization for Business Intelligence Systems*, Morgan Kaufmann Publishers, 2012.
[11] R.F. van der Lans, *Introduction to SQL; Mastering the Relational Database Language*, fourth edition, Addison-Wesley, 2007.