Master Data Management (MDM) Hub Architecture

75 out of 77 rated this helpful

Roger Wolter  
Microsoft Corporation

April 2007

Applies to:  
   Master Data Management (MDM)

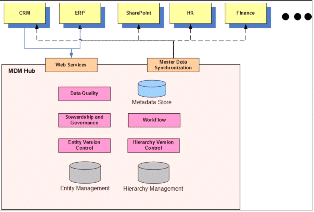
**Summary:** The second in a series on Master Data Management (MDM), this paper covers the architectural issues in building an MDM hub: a database with the software to manage the master data that is stored in the database and keep it synchronized with the transactional systems that use the master data. (17 printed pages)

Contents

[Introduction](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic1)  
[Hub Architecture Styles](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic2)  
[Versions and Hierarchies](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic3)  
[Population and Synchronization](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic4)  
[Metadata](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic5)  
[Stewardship and Governance](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic6)  
[Data Profiling](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic7)  
[Export](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic8)  
[Reporting](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic9)  
[Workflow and Business Rules](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic10)  
[Tools](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic11)  
[Conclusion](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic12)  
[Additional Reading](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic13)

Introduction

This paper is the second in a series on Master Data Management (MDM). If you have read the [first paper](http://msdn2.microsoft.com/en-us/architecture/bb190163.aspx), you know what master data is and the benefits of managing the master data in your organization. This paper continues that by covering the architectural issues in building an MDM hub. The MDM hub is a database with the software to manage the master data that is stored in the database and keep it synchronized with the transactional systems that use the master data. Figure 1 illustrates the architecture of a typical MDM hub.

[](http://msdn.microsoft.com/en-us/library/Bb410798.mdmhubarch01l(l=en-us).gif)

**Figure 1. MDM hub architecture (Click on the picture for a larger image)**

The MDM hub contains the functions and tools required to keep the MDM entities and hierarchies consistent and accurate. In this architecture, the MDM data can be accessed through a Web services interface. The Master Data Synchronization function is responsible for keeping the data in the hub synchronized with the data in the transactional systems (depicted across the top in Figure 1). There are several alternative implementation styles used for MDM hubs. The next section describes three of the more commonly used styles.

Hub Architecture Styles

There are three basic styles of architecture used for MDM hubs: the registry, the repository, and the hybrid approach. The hybrid approach is really a continuum of approaches between the two extremes of registry and repository, so I'll spend more time on the two extremes.

Repository

In the repository approach, the complete collection of master data for an enterprise is stored in a single database. The repository data model must include all the attributes required by all the applications that use the master data. The applications that consume, create, or maintain master data are all modified to use the master data in the hub, instead of the master data previously maintained in the application database. For example, the Order Entry and CRM applications would be modified to use the same set of customer tables in the master-data hub, instead of their own data stores. The advantages of this approach are pretty obvious. There are no issues with keeping multiple versions of the same customer record in multiple applications synchronized, because all the applications use the same record. There is less chance of duplicate records because there is only one set of data, so duplicates are relatively easy to detect. However, they obviously are not impossible, because things like alternate spelling, nicknames, multiple locations for the same company, typos, and so on are still possible, and the MDM hub must be designed to deal with them.

While the repository approach has significant advantages for maintaining a continuously consistent source of master data, there are major issues that must be considered when designing a repository-based MDM hub:

* The most obvious issue is that it's not always easy or even possible to change your existing applications to use the new master data. If you do not own the source for the application, you may not be able to modify it to use the new master-data hub. If the application's data model is pretty close to the MDM hub's data model, you may be able to use views and linked servers to make your application think it is talking to its own data, when in fact it is talking to the MDM hub.

I have also seen some systems that reduce the number of changes required in the applications by creating a stand-alone application that does some of the maintenance of the master data, so that not all of the application functionality needs to be ported to use the hub data. This approach is generally hard to implement in a way that users accept, however. Adding customers in a different application than the one used for updates is probably unacceptably complex. On the other hand, one of the more common reasons for implementing and MDM hub is to provide clean, consistent data for a SOA implementation. If you are rewriting and wrapping your applications as services, it might not be unreasonable to create new services to manage the master data.

* Another issue that must be resolved when implementing a repository-style MDM hub is coming up with a data model that includes all the necessary data, without it being so large that it's impossible to use. Because the hub database is used by all applications in the repository model, it has to include all the information required for all the applications. The simple answer to this is to make the hub database a superset of all the application data models. In this approach, a hub customer record would include all the attributes of the customer records of all the applications using the MDM hub. This is not practical, because it ignores many of the problems you need an MDM solution to solve. For example, if there are five formats for addresses, eight formats for telephone numbers, and six different customer IDs, making all of these columns in the customer MDM database would make the MDM hub almost unusable. Every query would have to decide which address, telephone number, and customer number to use. In many records, only one or two formats would be populated.

The obvious solution to this is to settle on an enterprise-wide standard for each of the data elements in the MDM hub and modify the applications to consume and produce the standard formats. This is not only a lot of work for the IT department, but determining whose format should become the standard format is often a major political problem. All the application owners think that their data formats are the right ones—not necessarily because the formats are any better, but because the application owners do not want to make the changes required to use a different format. It's not unusual for meetings held to settle on a data model to take as much time as the actual implementation of the project. If there are data elements that are used by only one application, the data-modeling effort might decide to eliminate them, and this might require significant changes to the application.

* Another significant data-modeling issue is what to do with data elements that are not used by all applications. For example, a customer added by an order-entry application would likely have significantly fewer attributes than a customer added by the CRM application. Or a product added by marketing might have attributes that are very different from a product added by engineering. In some cases, it might make sense to assign default values to unpopulated attributes; and, in other cases, you might decide to modify the application to populate the extra attributes. In an SOA implementation, you may decide to populate all the attributes with the service program. In general, there will be cases in which it is not desirable or possible to populate all of the attributes from all the applications. A typical example is the Product Information Management (PIM) part of an MDM system, in which it may not make sense to maintain the same attributes for a product that is purchased for resale as for a product that is manufactured in-house.

Registry

The registry approach is the opposite of the repository approach, because none of the master-data records is stored in the MDM hub. The master data is maintained in the application databases, and the MDM hub contains lists of keys that can be used to find all the related records for a particular master-data item. For example, if there are records for a particular customer in the CRM, Order Entry, and Customer Service databases, the MDM hub would contain a mapping of the keys for these three records to a common key.

Because each application maintains its own data, the changes to application code to implement this model are usually minimal, and current application users generally do not need to be aware of the MDM system. The downside of this model is that every query against MDM data is a distributed query across all the entries for the desired data in all the application databases. If the query is going against a particular customer, this is probably not an unreasonable query. But if you want a list of all customers who have ordered a particular product in the last six months, you may need to do a distributed join across tables from 5 or even 10 databases. Doing this kind of large, distributed query efficiently is pretty difficult. This is the realm of Enterprise Information Integration (EII). So, unless your requirements are relatively simple, you may want to look at EII-distributed query tools to implement query processing in a registry-model MDM hub.

There are basically two styles of repository databases used for MDM. The first has one row in a table for each master-data entity and columns for the keys of the application systems. This is the most straightforward to implement and the most efficient in operation, because all of the distributed queries for a given MDM record can start from the same database row. A NULL value for a particular key means that the corresponding database does not contain a record for the given MDM entity.

There are two significant issues with this scheme, however. First, adding an application to the MDM hub means adding columns to the key-matching table, which is not a big issue, but it may also mean changing queries to include the new source of information. The second, more significant issue is that this style assumes that a given database has only one record for a given MDM entity. While this would be ideal, it is rare to find this in a real application. One obvious solution to this is first to clean up the application databases, so there *is* only one record for each master-data item. This should be one of the goals of any MDM project, but it's not always possible to make the database cleanup a prerequisite for including an application in the MDM hub. If it is impractical to clean up the application database before integrating it into the MDM hub, the repository can be designed with one row for each mapping from the MDM entity to an application record. For example, if Ford has 20 records in the CRM database, the MDM hub would have 20 rows mapping the Ford MDM identity to each of the different CRM customer numbers. This style makes for much more complex queries and also raises issues, such as how to deal with 10 different addresses for the same customer. Nevertheless, it might be a necessary step in the evolution of your MDM solution. Knowing that there are 20 CRM records for Ford is a necessary first step in consolidating them into a single record.

Hybrid Model

As the name implies, the hybrid model includes features of both the repository and registry models. It recognizes that, in most cases, it is not practical (in the short term, at least) to modify all applications to use a single version of the master data, and also that making every MDM hub query a distributed query is very complex and probably will not provide acceptable performance. The hybrid model leaves the master-data records in the application databases and maintains keys in the MDM hub, as the registry model does. But it also replicates the most important attributes for each master entity in the MDM hub, so that a significant number of MDM queries can be satisfied directly from the hub database, and only queries that reference less-common attributes have to reference the application database.

While at first it seems that the hybrid model has the advantages of both of the other models, it is important to note that it has issues that neither of the other models has. Only the hybrid model includes replicated data (other than keys), so only the hybrid model must deal with update conflicts and replication-latency issues. The hybrid model also has the same data-model issues that the repository model has. Which attributes are stored in the hub, what they are called, and what format they are in can be very contentious issues when the hub integrates data from many disparate systems.

Architectural Issues

The following is a brief discussion of some of the architectural issues that must be considered in the design of an MDM hub database.

Data Model

In all three models, the design process must include a common data model for the hub database. In the repository model, the MDM data model becomes the hub-database data model. The model includes mapping from the application data models to the MDM data model, but these mappings are used only to create the hub database and define the application changes required to modify the application to use the hub database as the source of their master data.

The other two hub models also require an MDM data model and mappings from the current applications, but they are used differently. In the registry model, the data model is used to define queries and views, and the mapping is used to do the necessary transformations to map the application data to the MDM data model in each query. In the hybrid model, the common attributes are replicated to the hub database and the non-common attributes are transformed as part of queries, so both kinds of mapping are used. Almost by definition, there will be alternate mappings for some attributes, and rules must be defined for which mapping to use. For example, a customer address is generally stored in several databases so rules must be defined to control which address to use first and which alternate to use if the preferred address isn't available. (These business rules can get to be pretty complex if many databases are integrated in the MDM hub, so I will talk about business rules later in [Workflow and Business Rules](http://msdn.microsoft.com/en-us/library/bb410798.aspx#mdmhubarch_topic10).) The data models and business rules are documented in the MDM metadata and should be used as required to implement data-driven processing for populating, maintaining, and querying the MDM hub data.

MDM Hub Model

We have covered the three hub-database models, so let's discuss how to decide which model to use. The repository model is the most attractive, because it provides a real source of master data that is always current and consistent. The other choices involve data replication, so there is usually some latency between data updates and hub updates. Master data is generally fairly static, so a little latency is not necessarily unacceptable. The non-repository approaches also maintain multiple copies of some data, so consistency (keeping the copies the same) is an issue these approaches must deal with.

The downside of the repository model is that it can be extremely expensive and take a long time to implement, because it requires changes to the applications that maintain and consume the master data. The repository model makes sense if: the number of applications involved in the MDM project is limited; you have enough control over the applications to make the required modifications; and the availability of authoritative and consistent master data provides enough business value to justify the time and cost required to build a repository-model MDM hub.

A registry-model MDM hub is appropriate when only a limited number of non-performance–critical queries involve access to a significant number of the application databases integrated with the MDM hub. Registry-model hubs are cheaper and quicker to implement and can be implemented one data source at a time, so they are good for incremental implementation and provide early return on investment (ROI). Registries are not good when queries routinely return attributes from many application databases or when there is enough duplication of data, so that determining which of several alternate sources of an attribute to return is a complex decision. In these cases, the pre-integrated and cleansed data provided by a hybrid-model MDM hub provide a more efficient and consistent source of master data.

It's important to note that the hybrid model is not a single model, but instead a whole continuum of options that start at the registry model and continue through to the repository model. For this reason, you may decide to start with a solution close to the registry model and gradually expand the number of attributes integrated into the MDM hub until you have an MDM repository implemented. Because MDM projects can be very expensive and time-consuming in a large enterprise with many applications, it's good to have a strategy that allows you to implement incrementally by both gradually increasing the number of attributes stored in the hub and incrementally adding applications to the hub. This allows you to show an early ROI from the MDM project, with a clear path to a long-term enterprise-wide solution.

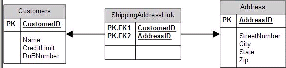
Versions and Hierarchies

The previous section explained the options for implementing an MDM hub. This section drills into that a bit by discussing versions and hierarchies—two features that are keys to an MDM hub implementation. It covers why they are important and presents a few implementation options.

Link Tables

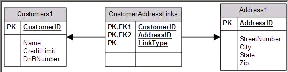
In the implementation options for both of these features, I refer to link tables frequently, so I thought I would explain what I mean when I say *link table*. (If you are already a link-table expert, feel free to skip to the next section.)

One of the fundamental concepts in relational databases is using a foreign key to define a relationship between related rows. This is done by storing they key of the related row in a column of the other row. For example, if I have a table of customers and another table of addresses, I can specify the shipping address for a customer by placing the primary key of the address table in a column named "shipping-address" in the customer table. When you want to find the shipping address for a customer, you use the value in the shipping-address column for that customer to look up the address. Many customers can use the same address by using the same key in their shipping-address column, but there's no good way to model a single customer with many shipping addresses. In reality, many customers can have the same address, and one customer can have many addresses. This is called a many-to-many relationship, and the easiest way to model this is with a link table. A link table looks something like Figure 2.

[](http://msdn.microsoft.com/en-us/library/Bb410798.mdmhubarch02l(l=en-us).gif)

**Figure 2. Link table (Click on the picture for a larger image)**

Another useful property of link tables is that columns in the link table can be used to represent properties of the relationship. For example, a relationship between customers and addresses might represent a shipping address for a customer or a billing address for a customer. You could represent this by having two different link tables—one for shipping addresses and one for billing addresses—or by having a single link table to link customers and addresses with a link-type column that is used to differentiate between shipping-address links and billing-address links, as described in Figure 3.

[](http://msdn.microsoft.com/en-us/library/Bb410798.mdmhubarch03l(l=en-us).gif)

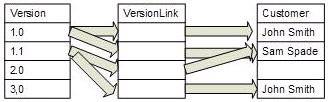
**Figure 3. Typed link table (Click on the picture for a larger image)**

Notice that all the information about the relationship is included in the link table. Neither of the tables that are tied together has any information about the link. This means you can create a new relationship between tables that are part of applications that can't be changed. For example, you can create a relationship between a customer record in the CRM application and a territory record in the Sales Force Automation application without changing either database.

Versioning

Data governance and regulatory compliance are much easier with a complete version history of all changes to the master data. It is often not enough to know what a customer's credit limit is today; you need to know what his credit limit was three months ago, when the customer was charged a high interest rate for exceeding his limit. While this is a simple example, there are many cases in which knowledge of past values for master-data attributes may be required. This leads to versioning as a key feature for master-data management systems. Versions are also required to support data stewardship and governance activities on master data. When master data is modified, business rules are applied to the modifications to determine if they meet the rules developed by the data-governance organization. Data stewards also use version information to monitor the results of the updates and, if necessary, restore the original values.

When most developers think of versioning, they picture source-code control systems that have full branching and merging capabilities. If your MDM hub needs this type of versioning, the versions are generally implemented with link tables that link rows in a version table with a particular version of the MDM record. A simplified diagram of the links might look something like Figure 4.



**Figure 4. Versions with a link table**

Notice that John Smith changed in version 1.1, so there are two different rows for John Smith; but Sam Spade did not change, so both versions point to the same row. In this schema, adding a new branch involves adding a row to the version table and creating rows in the VersionLink table for every customer. As customers are updated, a new row is inserted for each modified customer row and the link is changed to point to the new row. While this method offers a lot of flexibility, millions of customers and hundreds of branches produce huge link tables, so managing the volume of data can be an issue. Also, even fairly simple queries like "select all customers with a past-due invoice" involve multiple joins to obtain the right version of the customer records. In my opinion, most MDM systems do not require this level of versioning flexibility, and trading reduced flexibility for simplicity and performance is a good option.

One of the simplest versioning schemes is to add an "EffectiveDate" column to each master-data row. When a master-data item is modified, a new copy of the row is inserted with the "EffectiveDate" column set to the date and time that the change was made. (Okay, maybe it should be "EffectiveDateTime.") When you want to query the latest version of all customers, you look for the MAX(EffectiveDate). If you want to know what a customer record looked like on a particular date, you look for the row with the maximum EffectiveDate in which the EffectiveDate is less than the date you are looking for.

This versioning scheme is another place in which link tables are a handy tool. If you model a customer shipping address as a foreign-key column in the customer row, changing the address requires adding a row to the address table for the new address and adding a new row to the customer table that contains the address key. With a link table, you just add the address-table row and a corresponding row in the link table. Whether this is a good thing or not depends on whether you consider a change to the shipping address to be a change to the customer. If you want to track the address change as a new version of the customer record, the foreign-key relationship is a better way to model the customer record, because each change to the address produces a new version of the customer row.

One of the downsides of maintaining a version history of all your master-data entities is that even simple queries have to deal with versions to retrieve the correct version of the data. One way to simplify this is to create a view that exposes the latest version of all objects, so that users who care only about the latest version can write simple queries and only users who need a particular version need to deal with the versioning complexity.

Another alternative solution that also may reduce the management overhead of the hub database is, instead of inserting a new row into the master-data table when a record is modified, to actually modify the master record in place and put the old version into a history table. This can make your master-data tables orders of magnitude smaller, in addition to making non-version queries simpler to write. Because the historical data is accessed less often than the latest version, it can be stored on slower, cheaper disks—reducing the overall cost of the system.

Another problem the history-table approach solves is what happens when the master-data schema changes. For example, when you add columns to the customer table, what value do you put into the new rows for old versions of the customer record that did not include the columns? Or, more importantly, if you drop a column, what happens to the information stored in older versions? With history tables, each schema version can be stored in a separate history table with the schema that was in use at the time the rows were created. This obviously makes queries against historical data more complex, because you will need to know which table contains the versions you want, but it provides a more accurate representation of history—another trade-off to consider.

The final option for representing versions is to use change records similar to the deltas maintained in a source-code control system. In this scheme, the current version is stored along with a log of the changes done to arrive at the current version. To obtain a past version, you start with the current version and undo the changes from the log until you arrive at the version you want. This is obviously much more complex than the previous options, but the total amount of data stored in this case is much less. You should not consider this model if you need to do a lot of queries against previous versions, because they can be very expensive. For example, obtaining a list of product prices for all products as of December 2 of two years ago would require rebuilding every customer from the change log.

Hierarchies

For purposes of this article, *hierarchy management* is defined as the ability to define and store relationships between master-data records in the MDM hub. Relationships are a critical part of the master data: Products are sold by salesmen, employees work for managers, companies have subsidiaries, sales territories contain customers, and products are made from parts. All these relationships make your master data more useful.

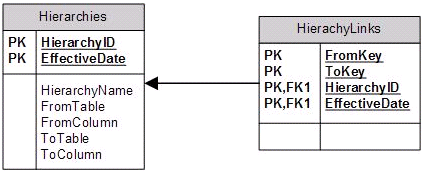
Many relationships exist in your current systems. For example, your HR system may track who works for whom or which organization pays your salary. Other relationships may be possible to define only because the MDM hub integrates the data from multiple systems. For example, linking a customer in the CRM system to a service contract in the customer-service system may be difficult to do if the systems are not aware of each other; but if both the customers and service contracts are stored in the MDM hub, a link table can be defined to track this relationship.

Some hierarchies are special-purpose or temporary. For example, if your development teams are organized in a matrix structure, expenses and salaries may be rolled-up to a management structure for budgeting and to a project structure for time and expense reporting.

MDM hierarchies should be named, discoverable, versioned, governed, and shared. For example, if I want to know how expenses for the XYZ project are rolled up or who reports to John Smith, I should be able to select the appropriate hierarchy from a list and know whether it is authoritative and when it took effect. This means that everyone who looks at project expenses will use the same structure, instead of everyone using whatever spreadsheet they happen to find. This also means that if an auditor wants to know who was working on the project on November 2, 2004, there is a single authoritative place to find the answer. CEOs love this stuff, because it tends to keep them out of jail.

To support relationships between entities without requiring changes to the entities, most hierarchies are implemented as link tables. If the data already contains relationships imported from the source systems, it generally makes sense to leave those relationships alone to maintain the fidelity between the MDM hub and the source system. But you may decide to convert them to hierarchies implemented as link tables to take advantage of the hierarchy-management features of the hub, as well as to provide a standard format for hierarchies.

Figure 5 shows a simplified view of what a hierarchy-management data model might look like.



**Figure 5. Hierarchy link table**

In reality, there would be quite a bit more metadata about the hierarchy and probably more properties on the link-table rows. Whether you implement all hierarchies in the same table or create a table for each hierarchy will depend on how uniform and how big your hierarchies are. One hierarchy per table is the correct way to model it, from a relational-theory standpoint; but if you have hundreds of fairly small hierarchies, combining them may simplify database maintenance. There are a number of intermediate options, too. For example, you may group all the hierarchies that use the same pair of keys into a single table or group them by use—accounting in one table, HR in another, and CRM in a third.

Population and Synchronization

You should now have a good understanding of the architectural issues around deciding what your MDM hub database looks like and what kind of data is kept in it. In this section, we will discuss how to populate the hub with good, clean data and how to ensure that the data stays clean and consistent. This involves populating the hub database with data from the source systems initially and—with the exception of a pure repository-model hub—keeping the source systems synchronized with the hub database as the source systems make changes to the data.

Batch Loading: ETL

The initial population of an MDM hub is very similar to populating the dimension tables in a relational data warehouse. In many cases, the same Extract, Transform, and Load (ETL) tools used for data-warehouse loading can be used to populate the MDM hub. Many MDM implementations use either standard ETL tools or tools derived from ETL tools. A typical load process involves the following steps:

1. **Extract the data from the source system.** This should probably be done one subject area at a time, to make things easier. This is that part of the process that may require either buying or building an adapter that understands the data source. Again, the same adapters that are used to extract dimension data for data warehouses should work here, unless you are using a tool that is not compatible with standard adapters. This is basically a batch operation, so many tools will extract into a flat file, while others will extract directly into the ETL pipeline.
2. **Transform to the hub data model.** As part of the hub-design process, a data model was defined along with a mapping from each source to the common hub model. This step in the process makes the necessary changes to transform the master-data entity from the application data model to the MDM hub data model. This again is standard ETL stuff that might include changing column names, changing field sizes, changing formats of things like telephone numbers and addresses to match the standard formats for the MDM hub, combining columns into a single column, and parsing a single column value into multiple columns.
3. **Check for duplicates.** This process is the "secret sauce" of most MDM systems. It is both the hardest and most important part of populating the MDM hub. If you want a single view of your customer or product data, records describing the same business entity must be combined into a unique record for each unique entity; but if your MDM system is too aggressive in finding duplicates, entities might disappear when they are incorrectly determined to be already in the system. For example, your duplicate-detection algorithm might decide that George W. Bush and George H. W. Bush are the same person, so information about one of them might be lost. This is one of the reasons that both versions of the record should be stored in the version history, so this kind of error can be corrected if necessary.

Some duplicate-checking algorithms are fairly simple and check for things like alternate spellings and missing words—for example, John Smith, Mr. John Smith, J. T. Smith, and so forth. While these are adequate for reasonably small databases, the potential for false matches is high. More-sophisticated algorithms might check for people at the same address or with the same telephone numbers. Other systems might use external data like telephone-directory data or Dun & Bradstreet listings to find matches. Many tools specialize in certain kinds of data—medical-patient data, consumer goods, or auto parts, for example. If there is a tool available for the kind of data you work with, these specialized tools can provide very accurate matching. Other tools are more generic and often allow you to specify your own matching rules to improve the matching for your specific data.

Almost all of the matching tools provide a "degree of confidence" number for each match they detect, and your loading process should specify what confidence level is required for a match. For example, you may decide that a 95 percent confidence level is enough to automatically match an entity, confidence levels between 80 percent and 95 percent should be marked for manual processing, and levels below 85 percent are not considered matches. What values you choose will depend on the consequences of a false match or a missed match. If the result of a mistake is sending two marketing brochures when one would have been adequate, the confidence level does not have to be high; but if a mistake results in someone getting arrested for tax evasion or treated for the wrong illness, it's good to be very sure.

1. **Load the MDM hub database.** If the new record is not already in the hub database, this is just a matter of inserting the data into the correct tables. But if it is a duplicate, the load process must check the business rules for this entity to decide what data to update with the incoming record. For example, if there is no shipping address in the current record and the incoming record includes a shipping address, the address is added. If there is already a shipping address and the incoming record also has one, there must be a rule specified to decide which one to keep or if both should be kept. If the business rules can't resolve the conflict, the incoming record should be put on a queue for manual processing. If the MDM hub is a registry or hybrid model, even if none of the data from the incoming record is used, the key of the record should be added to the database to record the connection from the hub record to the source record. This may be used by queries to find the source record or by the hub to publish hub updates to the source systems. See the next section for more on this.
2. **Update the source systems.** If loading a new record changes the hub database, the change may need to be propagated to one or more of the source systems. For example, if a new, authoritative shipping address is added to a customer record, other applications that stored information about that customer may want to use the new address. I say *may*, because there are cases where an application needs to continue with the old address and ignore the new address. I will cover this process in more detail in the synchronization discussion, but I just wanted to mention it here for completeness. As I said at the beginning of this section, if your MDM hub uses the repository model, it will replace the databases in the source systems, so this step is unnecessary.

The process of loading the data from a source application into the MDM hub can take a long time, if there is a lot of data and if a significant amount of manual processing is required to resolve data-quality issues. In many cases, it is wise to load a source application into the hub and then run for a few days or weeks to ensure everything is working correctly before loading the next application. The load process works best if the most authoritative and complete data sources are loaded first, so that subsequent loads make relatively few changes to the existing hub data. Primarily, however, it's best to record duplicates and synchronize the application data with the hub data. Loading the most critical databases first also leads to earlier time to value, which can be important in justifying the MDM investment.

Synchronization

Now that the MDM hub is populated with a single authoritative version of your master data, you need to develop a process to keep it clean and authoritative. This means implementing a method for changes to existing data and new master-data items to be transferred to the MDM hub, while maintaining the same level of data cleanliness that you achieved while loading the hub from the source applications.

One way of maintaining the MDM hub database is to keep any of the source applications from making changes to the master-data entities and thus force all additions and updates to the master data to be done to the hub database. This is the easiest technique to implement and manage, because only one database is updated and all updates can be closely monitored and controlled to ensure conformance to business rules. The primary difficulty with implementing this technique for maintaining the master data is that it requires that none of the source applications make updates to the master data. For example, nobody can add a customer to the CRM system and nobody can change a product definition in the ERP system. All changes must go through the new MDM system.

In many organizations, the retraining and operational changes required to make this work are unpalatable. On the other hand, if this MDM project is part of an SOA initiative, implementing new services to manage the master data can be incorporated into the overall SOA project. I will not spend a lot of time on how to build this service, because it is generally a pretty basic data-maintenance service. If you have access to the source systems, you might want to use a modified version of the best master-data maintenance procedures you currently have or, at least, use the business rules and validation logic from the source systems.

The one thing to remember here is that having a single master database does not mean you do not have to worry about duplicates. It's still possible for a user to create a new entity instead of modifying an existing one (and, in some systems, it is actually easer to create a new entry than to find and modify an existing one), so the MDM hub service must still check for duplicate entries.

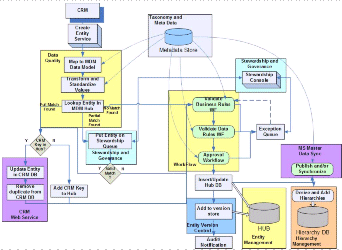
If moving all master-data maintenance to the MDM hub is technically or organizationally impossible, you can consider a synchronization process that transfers changed master-data records from the source application that made the change to the MDM hub. The MDM hub then processes the change using much the same logic that was used originally to populate the hub. This introduces the possibility of conflicting updates and inserts from multiple systems, and it introduces some latency between the time a change is made and when it shows up in the MDM hub database; so the business must understand the limitations of this system.

In most systems, the rate of change to a given master-data entity is fairly low, so update conflicts should be pretty rare and thus reasonable to resolve either manually or with simple business rules. This is especially true for data attributes that represent real-world entities. For example, the chances of two conflicting changes to a customer's telephone number or address happening the same day are pretty remote. To further reduce the chances of update conflicts, you might introduce the concept of a preferred source for data. For example, if it's not feasible to change the product-information–maintenance process to use a new service for maintaining product data, it may still be possible to limit the maintenance of any given product to a single system. This eliminates update conflicts, without requiring a total revamping of the product-maintenance process.

The most significant technical challenge in transferring master-data changes from the source applications to the MDM hub is detecting changes in the source system. If you have access to the source system, you may be able to add a little logic to send each master-data change to the MDM hub as it is made to the application database. Another option is to use database triggers to detect changes, if you have enough understanding of and control over the application database to do this. Replication might also be a good alternative, if the entities are simple enough that you can determine what the entity change was from the replicated data.

Unfortunately, you may find that none of these options works in your situation, so you might have to resort to periodically querying the application for changes or even parsing audit logs to find changes. After you have detected a change in the source system, it should be sent to the MDM hub as quickly as possible to reduce the update latency. I generally recommend reliable messaging for this task, to ensure that changes are not lost in network or system failures. Microsoft BizTalk and Microsoft SQL Server 2005 Service Broker are probably the best alternative for this on the Microsoft platform; but because the source applications can be running on a variety of platforms, other alternatives may be appropriate. On the other hand, if you are using the MDM Hub primarily for reporting and hierarchy management in a business-information (BI) environment, latency might not be a big issue; so loading changes into the hub database with batch-oriented MDM tools will provide adequate data freshness, with significantly less overhead and complexity.

After the change has been transferred to the MDM hub, it must be applied to the hub database. Figure 6 illustrates a typical process.

[](http://msdn.microsoft.com/en-us/library/Bb410798.mdmhubarch06l(l=en-us).gif)

**Figure 6. CreateEntity service (Click on the picture for a larger image)**

Figure 6 shows a CRM application adding a customer to the MDM hub by calling the CreateEntity service. Follow along with the flow, while I walk through the processing steps:

1. The incoming data is mapped to the MDM data model using the same transformations used in the ETL process described earlier. This makes checking for duplication easier and puts the record into a common format that can be used throughout the rest of the process.
2. The hub looks up the entity in the hub database to see if it is already there. This is not a simple SQL query; it does all the fuzzy-matching logic that the duplicate-elimination process did when creating the hub database. For this reason, it's good to look for a tool that can look for duplicates in batch mode and also do the lookup one entity at a time. As I explained in the ETL section, there are three possible outcomes of the search: duplicate entry found, no entry found, and do not know. If the answer is do not know, the entity is put on a queue for the data steward to resolve (stewardship will be covered in a later section).
3. If a duplicate is found, another application has already added this entity, so this insert will be changed to an update. The entity in the hub is checked to see if there is already an entry from the CRM application for this entity. If there is, this entry is a duplicate in the CRM database; so the entity already in the CRM database is updated with the new data, and the entity that the CRM application is trying to add will be deleted to eliminate the duplication. On the other hand, if the entity in the MDM hub does not currently have a key for the CRM application, the key for the incoming entity is added to the hub entity, and the incoming entity is passed on as an update to the approval workflow.
4. If no entry was found in the MDM hub for the incoming entity, it is passed to the approval workflow as an insert. At this point, the three streams converge again, and an automated workflow checks the data update or insert to verify that it meets all the business rules for the MDM hub, as defined by the data-governance activity of the enterprise. Some examples of business rules might be which fields require values, allowable value ranges, address verified with an external vendor, Dun & Bradstreet (D&B) number valid for this business, and prices in the correct currency. At some point, if there is not enough information to determine if a rule is satisfied, or a rule determines that manual approval is needed, the entity will be placed on the stewardship queue for manual approval.
5. If the entity passes the approval-workflow process, it is inserted or updated in the hub database as required. In the same transaction, the version information for this entity is updated with the previous values. If the entity contains information that can be used to derive any of the managed hierarchies for this record, the required entries are inserted in the hierarchy database. For example, the customer entity may be linked to a support-contract entity and a sales-territory entity based on **contract-id** and **territory-id** fields in the customer entity.
6. When the entity has been added to the hub database, the changes are published out to the other source systems. In some MDM systems, this means publishing every change to every system; but, in most cases, only a subset of the source systems are interested in changes to a given entity. For example, if your e-commerce system has hundreds of millions of customers, it probably does not make sense to push them all into the CRM system. A set of business rules can be used to determine which applications receive new master-data updates, based on entity type or source. Another alternative is to send updates to data only if the source application has a key in the MDM hub. In that way, an entity has to be added to the application before it is managed by the MDM system. The various methods of publishing updates back to the source applications are described in the next section.

Publishing Updates

The first architectural decision you must make about publishing updates is whether you need to do it. Some MDM systems are used to provide a single source of master data for enterprise reporting or performance management and do not require all the source applications to use the new master data. In many organizations, the political ramifications of an MDM system directly updating one of the key enterprise applications will prevent automatic propagation of updates to some systems. On the other hand, creating a clean source of master data is a significant effort, so it seems like a waste of resources not to propagate the cleaned-up data to all the source applications.

If you determine that you need to publish master-data updates, the next decision is whether to push updates out to the source application or let the source applications pull the changes from the hub. Pull is generally easier to implement and manage, but push reduces the time between when the hub is updated and the updates are available in the source applications. Pull is also generally easier to implement between heterogeneous systems. If your MDM hub runs on SQL Server and one of the source systems is on a mainframe, it will probably be much easier to have the mainframe read a change file than to write an application to push changes from Windows into the mainframe application. This is the classic trade-off of capability against complexity, and the deciding factors are usually the requirement of up-to-date master data weighed against the difficulty of doing the integration.

The push option looks like replication on the surface; and, in some cases, replication may be the best way to push the changes. This works if the source application data model is pretty close to the MDM hub data model and there is a replication-connection available. If the two data models are significantly different, if replication is not available between the databases, or if directly updating the source application's database is not allowed, an application-integration solution like BizTalk is probably the best choice. If necessary, this can include complex data transformations and even an orchestration to do the update in multiple steps. Orchestration can also be used to publish updates selectively to only the applications that require them. For example, only CRM systems that contain a record for a customer would receive updates for that customer. If you are publishing from one SQL Server database to another SQL Server database, Service Broker is also a good choice for a reliable asynchronous connection and transactional application of the required changes.

If the effort and complexity of implementing and maintaining a push solution are excessive, you may have to implement a pull solution. The simplest pull solution is to allow the application to query the MDM hub database (or preferably read-only views of the database) directly, to obtain the required data. If the amount of master data is pretty small, the application can periodically refresh its master data completely; but, in most cases, the application will want to refresh only what has changed. Time-stamp columns are the most common approach to this issue. Each application keeps track of the last time stamp it has on read-only retrieves data with time stamps greater than its remembered value. The downside of pulling data directly from the database is that, if it is done frequently by a large number of applications, it can cause significant performance degradation.

A pull alternative that makes it easy for applications to apply changes and reduces the load on the MDM hub database is to write changes into a journal or log. This can be either a database table or a flat file. If updates are sequentially numbered, an application can track which updates it has processed already. If the number of applications pulling data is relatively small, it might make sense to generate a separate journal for each application*.*This can be a lot of extra I/O, but it makes it easier to manage if each application can manage its own journal—by deleting the records they have processed, for example. On the other hand, you may want to maintain a journal of changes for auditing purposes, anyway, so this journal can do double duty. In a pull architecture, the application might pull updates itself or use an external tool that is either custom-written or implemented with an ETL tool to periodically read the changes and apply them to the source application. Some databases have Change Data Capture features that record all the changes to a specified set of tables in a file or table, so that a pull system can periodically read the captured changes instead of trying to determine what has changed.

It's also possible to do both push and pull, if your application requires it. For example, one of the destinations to which you push updates might be a service that writes a journal to support pull applications.

Data Integrity and Reliability

Something you must keep in mind when designing an MDM infrastructure is that the complex processing that happens in the background to support master-data synchronization must be extremely reliable. Losing updates will reduce the accuracy of the master data and can cause users to lose confidence in it. This means that, as you design the code that will handle data movement and manipulation, you must ensure that every action that changes or moves data is transactional and recoverable. Messaging should always be transactional to ensure messages don't get lost or duplicated; and all the asynchronous processing involved in the MDM workflows should be managed by transactional queues, so that it can be restarted from the previous state if the system shuts down in the middle of processing. Obviously, if I were designing the hub, it would use Service Broker to control its operations, but I'm sure BizTalk would provide the same reliability and would make the workflow and business rules much easier to implement. There are also transactional messaging environments available in non-Microsoft environments. The point is not which tool you use, but instead making sure that, as you go through every step of the process design, you consider what happens if the system shuts down unexpectedly or a disk drive fails.

Metadata

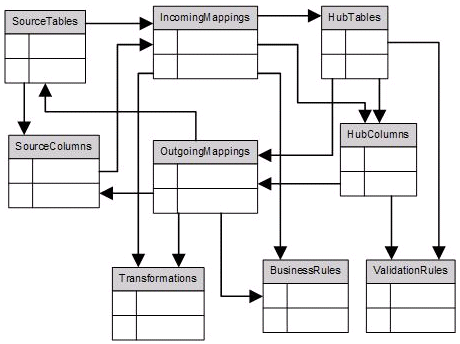
Metadata, as I'm sure you have often heard, is data about data. In MDM systems, as in any data-integration system, metadata is critical to success. As a minimum, for every column of every table in the MDM hub, there must be accurate data about where the value came from, what transformations were performed to get it into the hub data-model format, what business rules were applied to it, and which applications receive updates to it. As with the master data itself, metadata is only useful if it is current and accurate.

For compliance reasons, you generally have to be able to prove that the metadata accurately describes the master data. The easiest way to furnish this proof is to show that the processes that handle the master data are derived directly from the metadata or vice versa. For example, the mappings and transformations done by the ETL tools might be driven directly from the metadata, or the ETL tools might populate the metadata repository whenever a transformation pipeline is created. Business rules should be taken directly from the metadata, whenever possible. This is especially important in MDM, because the business rules often determine which of several alternate values is used to populate a data element.

Reading business rules and transformations directly from the metadata repository whenever they are used can be a performance issue, so either the information is cached or the actual data-maintenance code is generated from the metadata. The MDM metadata may also include descriptions of processing done outside the MDM hub. For example, the transformations required to map the hub data model back to the source systems that subscribe to it will probably execute on the source system, but must be defined in MDM metadata.

Because I have used the term several times, I should probably talk about metadata repositories. As the name implies, a *metadata repository* stores and manages metadata. Most enterprises that have a Data Administration group already have a metadata repository loaded up with information about all the databases and systems in the enterprise. This can be an invaluable resource for understanding the data models of the master-data source applications and which applications maintain which data elements in the database. A careful analysis of this metadata will go a long way toward determining what the MDM hub data model should look like. If you have a repository, the MDM metadata should be stored there also, and the relationship facilities of the repository should be used to record the mapping between each of the source-data stores and the MDM hub data model. This information is required both to implement the loading and synchronization processes and to document the derivation of each data element of the master data.

If you do not have a repository and do not have the time or budget to buy and populate one for the MDM project, you can consider storing the metadata in normal database tables. A useful starting point for what your metadata should look like are the standard Information Schema descriptions provided in many database systems. For example, look at the **INFORMATION\_SCHEMA** views in the SQL Server database. You can tie the source-data definitions to the hub definitions with link tables, as I described previously. These links can be tied to the transformations and business rules used to map the source to the hub schema. Figure 7 is a much-simplified depiction of what the data model for a metadata repository might look like.



**Figure 7. Metadata-repository data model**

The mappings tables are link tables that tie the source-data definitions to the hub-data definitions. I did not include any attributes for the entities, because I wanted to keep things simple. Most of the modern ETL and business-rules tools store their configuration and logic as XML files, so many of the attributes will be XML columns. Building a metadata repository can be a daunting task, so start simple and add features as you need them. In a previous life, I spent five or six years building and using metadata repositories, so I'll probably do a separate article on repositories after this current series is complete.

Stewardship and Governance

I hesitate to talk about stewardship and governance together, because they are different things that often get confused with each other. But there is a close relationship between them, so I will try to explain the differences.

In my simplified view, *data governance* is the process of defining the rules that data has to follow, and *data stewardship* makes sure that the data follows those rules. You might compare this to the legislature that makes the laws and the police who enforce them, but be careful not to carry the analogy too far. Poor-quality master data can affect many core applications of a business, so governance and stewardship are important functions in an MDM process.

The increasingly strict regulatory-compliance environment of the post-Enron world has made data governance a necessity for many companies. It's not enough to have the numbers; you have to be able to show that the numbers are based on accurate and verifiable data. This means both a governance function to demonstrate that the right controls are in place, and a stewardship function to ensure that the controls are enforced.

Governance is a huge topic, so I will just give a few examples of things governance controls. The governance rules should include who can read, create, update, and delete data; what validity checks are required for data; which application is the preferred source for data items; how long data is retained; what privacy policies are enforced; how confidential data is protected; and what disaster-recovery provisions are required, to name a few. The data-governance function should include leaders of both the IT and business groups of a company. This partnership used to be a hard thing to achieve; but with CEOs going to jail, and data on stolen laptops making front-page news, management is paying a lot more attention.

The role of a data steward is to ensure that the master data is clean, consistent, and accurate. In some of the earlier sections of this paper, I talked about cases in which data quality could not be determined with automated rules and workflows, so manual intervention was required. The people doing this manual intervention are the data stewards. There have been long-running debates over whether a data steward should come from the business or IT side of the house. I personally think the right person to be a steward for a particular collection of master data is the person who understands the data the best. In many cases, only someone who understands the business can make the tough calls about which value is correct for a particular data element. But in most companies, getting business experts to do what they see as the job of IT is difficult, at best.

The technical aspects of data stewardship involve a set of tools that help a steward find, analyze, and fix data-quality issues. These tools are generally integrated into a "stewardship console" that incorporates the data-profiling, data-analysis, and data-modification tools into a single user interface (UI). If your data stewards are business people, the stewardship console should be simple and highly automated. In organizations with complex governance rules and approval processes, workflow can be a useful part of the stewardship console. Basically, master-data updates that cannot be approved automatically are placed on a queue for the appropriate steward to resolve. An automation-supported human workflow can handle the routing and approval processes for these changes.

Data Profiling

Data-profiling tools can scan data for violations of business rules, missing values, incorrect values, duplicate records, and other data-quality issues. Profiling the data in your source systems is a good place to start an MDM project, so you can find out how much trouble you are in. Profiling can help you choose the authoritative source for data and design the ETL logic required to clean up and load the data into the MDM hub. Profiling should also be done periodically after the MDM system is in place, to find data-quality issues that the MDM system is not fixing.

Export

As soon as you have a clean, accurate source for master data, you will need to be able to export it to other systems that need it. For example, you may need to export your product master data periodically to be used in a data pool or a marketing campaign. This is not rocket science, and most databases include tools for exporting data in a given format or XML schema, so I just mention it here because your project should include a plan for doing exports.

Reporting

This includes reports of the master data itself—customer lists, product information, org charts, and so on—and reports on the health of the MDM hub itself. Things like the number of rules violations detected, the number of manual interventions required, and the average latency of master-data updates will help the IT organization discover issues early enough to prevent major problems. A solid reporting system that produces "canned" reports and allows user to design their own reports is an important part of the MDM hub.

Workflow and Business Rules

I think I have already covered all the places I can think of in which workflow automation and a good business-rules engine are required. The point I want to make here is that a good rules engine is critical to success in an MDM hub. In many cases, the rules are established by relatively unsophisticated data stewards, so a simple wizard or UI for developing rules may be required. The rules will often involve retrieving and manipulating database data, so a rules engine that has good database integration would also be useful.

Both automated workflow and human workflow are required, so a workflow engine that handles both would be useful. In a Microsoft environment, the Microsoft BizTalk workflow and Business-Rules Engine (BRE) are probably the closest fit. You might also consider Windows Workflow Foundation and Microsoft Windows SharePoint Services workflow, but their rules support is not as complete as in BRE.

Tools

An MDM project will probably need data-modeling tools for recording the data models for the source applications and the MDM hub. If you have a repository, the modeling tool should integrate with the repository. Other than that, most quality modeling tools will work; the requirements are pretty minimal. ETL tools for loading the hub data, data-quality tools, profiling tools, and application-integration tools are required for loading and synchronization. If your hub uses a registry or hybrid model, a distributed query tool may be needed for queries against the master entities when some parts of the data are stored in the source systems. A tool for defining and maintaining hierarchies is required for hierarchy management. I have already talked about the need for a simple, user-friendly stewardship console. Be sure it has a good way to browse versions and hierarchies in addition to the entity-management capabilities.

Conclusion

If you learned nothing else from this article, I hope you understand that MDM is both a pretty straightforward application of technologies that have been around a while and a fairly massive undertaking in most organizations. If your organization already has a data-administration function or at least some reasonably competent data modelers and data architects, you probably have the skills you need to succeed. Most large organizations have massive amounts of master data, so getting it all under control will take a while. Start with something that has limited scope, but also something that has a lot of value to the organization. Early success is important not only to get management to continue the project, but also for the project team to gain the satisfaction and confidence that deploying something that has significant impact brings. As soon as you have one part of the problem solved, learn from the experience, and repeat the process as many times as necessary to complete the MDM process.

Additional Reading

Articles

* [Data-Integration Solutions for Master Data Management](http://msdn2.microsoft.com/en-us/library/aa964123.aspx)
* [The What, Why, and How of Master Data Management](http://msdn2.microsoft.com/en-us/architecture/bb190163.aspx)
* [Fuzzy Lookups and Groupings Provide Powerful Data-Cleansing Capabilities](http://msdn.microsoft.com/msdnmag/issues/05/09/sqlserver2005/default.aspx)
* [Fuzzy Lookup and Fuzzy Grouping in SQL Server Integration Services 2005](http://msdn2.microsoft.com/en-us/library/ms345128.aspx)
* [Data-Quality Solutions](http://msdn2.microsoft.com/en-us/library/aa964137.aspx)
* [SQL Server 2005 Business Intelligence Metadata White Paper](http://www.microsoft.com/downloads/details.aspx?familyid=182bd330-0189-450c-a2fe-df5c132d9da9&displaylang=en)

Donald Farmer's blogs

* [BeyeBLOGS.com](http://www.beyeblogs.com/donaldfarmer/)
* [Donald Farmer on SQL Server Integration Services](http://sqljunkies.com/WebLog/donald_farmer/default.aspx)

My blog (Have Data Will Travel)

* [MDM and EAG and CDI Oh My! (Part 1)](http://blogs.msdn.com/rogerwolterblog/archive/2006/05/08/592749.aspx)
* [MDM and EAG and CDI Oh My! (Part 2)](http://blogs.msdn.com/rogerwolterblog/archive/2006/05/12/596715.aspx)

And a whole bunch more

* [http://search.msn.com/results.aspx?q=MDM+or+CDI+or+PIM+data+management&mkt=en-US&form=QBRE&go.x=6&go.y=5](http://search.msn.com/results.aspx?q=MDM%20or%20CDI%20or%20PIM%20data%20management&mkt=en-US&form=QBRE&go.x=6&go.y=5)