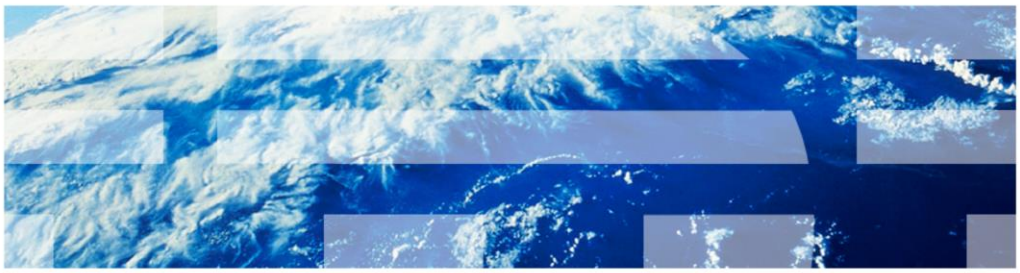
IBM

IBM Initiate

Understanding Master Data Engine searching



This is the “Understanding Master Data Engine searching” training presentation from IBM Initiate®. IBM Initiate Master Data Service® is referred to as MDS throughout this presentation.

Training objectives

- IBM Master Data Engine searches mechanism
- Benefits of preprocessing data
- Brief overview of phonetic conversion process
- Search examples
- Troubleshooting: Learn about potential problems
- Where to find more information

2 Understanding Master Data Engine searching

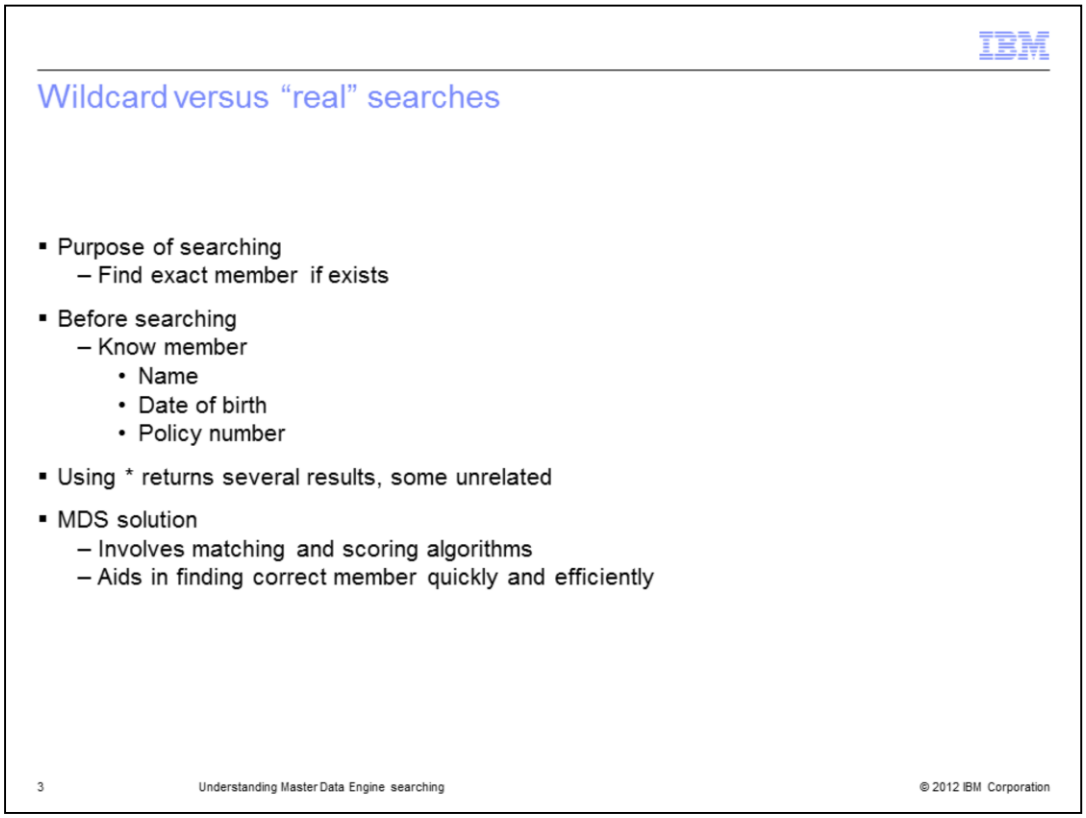
The objective of this presentation is to introduce you to the inner workings of the IBM Initiate Master Data Engine searching system. You will learn the reasons why the IBM Initiate MDS searches are more effective than those that use ‘wildcard’ searches. You will understand how the preprocessing of data increases search efficiency and improves data match-ability of members that are present in the system.

The phonetic conversion process is also discussed. This section covers how the MDS phonetic algorithms are able to retrieve members whose names sound similar to others. Even if the name is misspelled when searched, the name can still be retrieved from the system. This is largely due to the employment of phonetics, which compensates for potential data entry errors.

This presentation displays an example on the creation of sample records. Described is a step by step process of how records are compared in the MDS engine. This includes the retrieval of potential candidates as a result of sample searches.

A troubleshooting section is provided to cover some of the potential issues that users frequently run into when searching for records. Some of the examples have some real world situations and demonstrate how these can best be handled when unexpected results are encountered.

Finally, you are given a set of additional resources where you can get further information regarding this essential functionality of the MDS software.



First on the list is a discussion on wildcard searches versus real searches.

The ultimate purpose of performing a search is to be able to retrieve the member that you are looking for. Many people have become accustomed with standard wildcard searching and expect to find a specific member by typing part of a name and including an asterisk sign. For example, you want to find the name Edward in a system. You will typically enter “ED*” for the search. This can potentially retrieve hundreds of results, including names like Edgar, Eddy, Eduardo, Edmonton, Edna, and so on. As you can see, in order for you to get the particular record you need, you will have to search again through your results. This is exactly what the software tries to facilitate for you.

Some important identifiers such as a date of birth, social security number or any other key identifiers should be entered to produce more unique and better results.

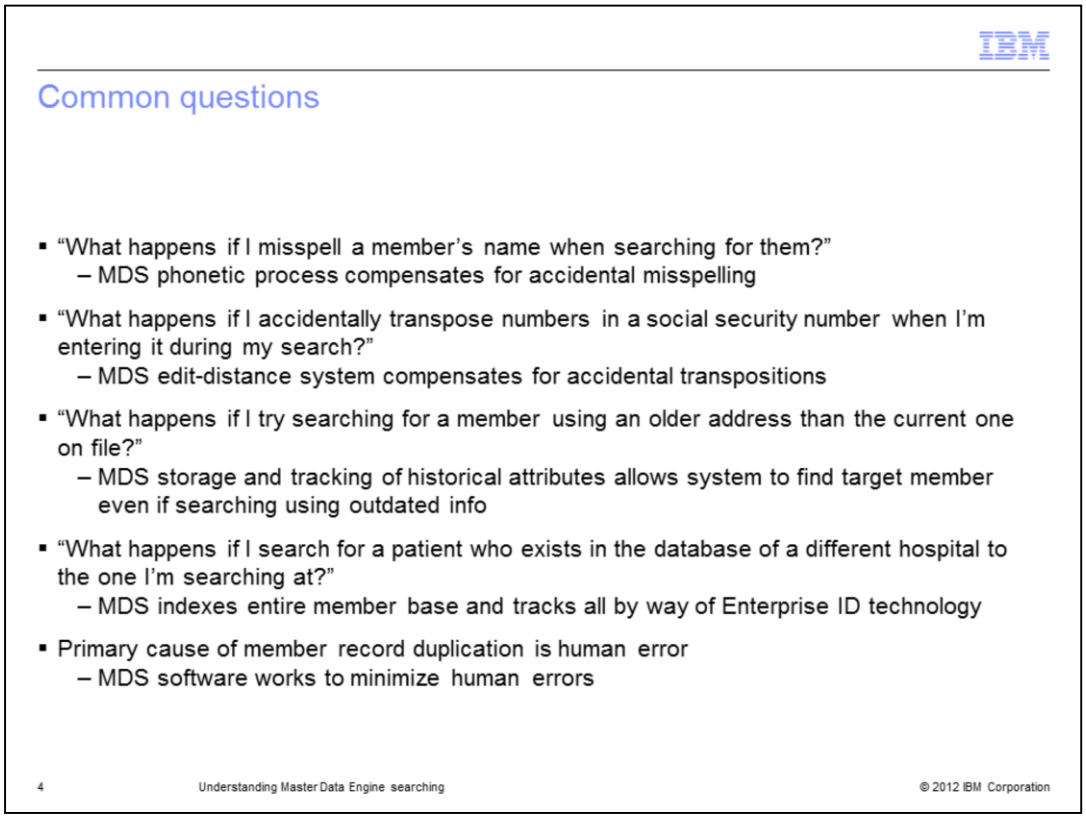
Suppose you are trying to find a particular member record with the name of “Patty Countryman”. A wildcard search on “PAT” will return: “PAT SMITH”, “PATTY JONES”, “PATRICK ADAMS” and “PATTY COUNTRYMAN”. As you can see, the records that have a textual match are returned along with Patty Countryman. However, they are nowhere close to the person you are looking for. “PAT SMITH” is a male whose last name is nothing like what you are looking for. “PATTY JONES” is a female, but her last name is also a mismatch. “PATRICK ADAMS” is also male, and does not match the record that you are interested in finding. This list can go on for hundreds of other records. You can see how wildcard searching can become cumbersome, especially if you have a spelling mistake to begin with. The MDS technology tries to avoid the returning of any arbitrary members that matched their name textually due to the use of a wildcard.

Suppose you are interested in finding a person whose social security number is 234-45-5678. If you were to search for 234*, the returned list of numbers that match those first three numeric characters will be extremely long, mostly filled with completely unrelated records. Again, wildcard searches are not useful in helping you retrieve a particular member.

The MDS solution involves the use of matching and scoring algorithms that aid you in finding the correct member quickly and efficiently. For instance, when searching for PATI COUNTRYMEN, spelled P A T I C O U N T R Y M E N, having misspelled both the first and last names, the system will deduce that PATTY COUNTRYMAN, spelled P A T TY C O U N T R Y M A N, is the actual member that needs to be retrieved, thanks to a phonetic match that these names have and despite a spelling mistake.

Searching for “234-54-5678”, where you have accidentally transposed the middle two numbers, will still get a match on the correct social security number as the edit distance is only one character.

There are several other helpful functions within the MDS software that allows you to find a member accurately.



This slide displays some commonly asked questions.

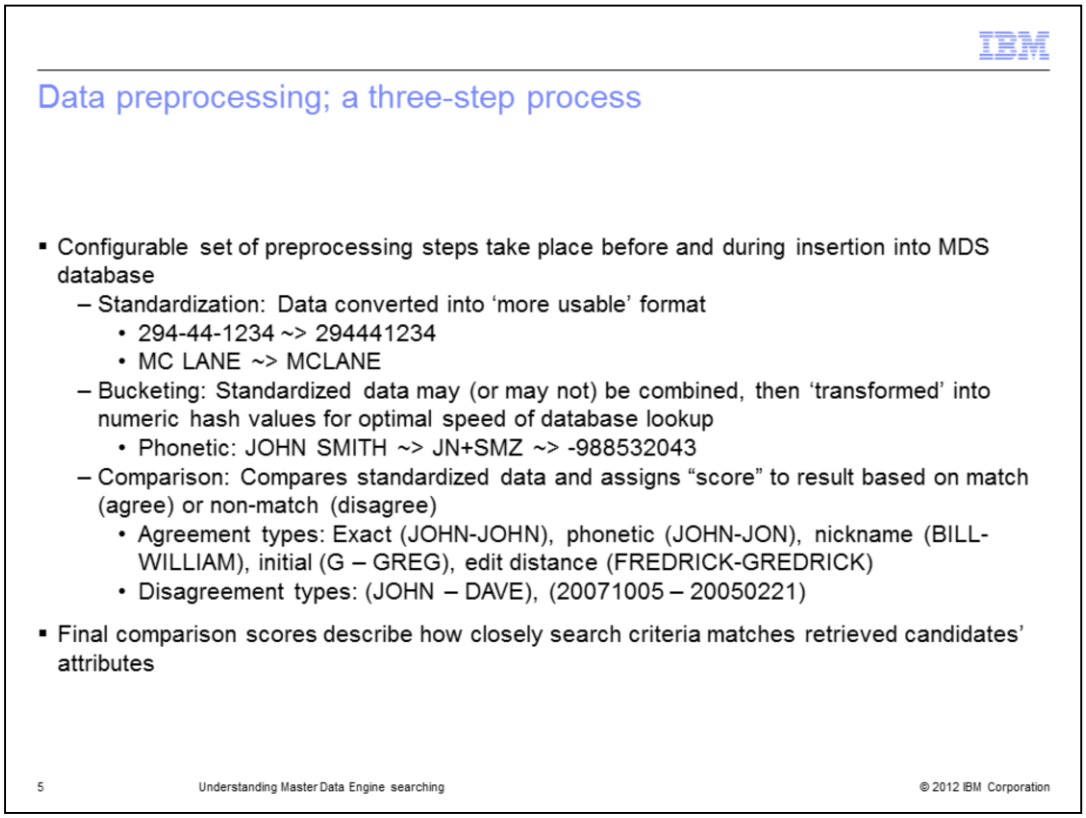
When asked, "what happens if a name is misspelled when doing a search?". The answer is that the MDS phonetic process will help in compensating for this mistake, helping you find the correct member.

When asked, "what happens if a search is done on a social security number whose numbers are accidentally transposed?". The answer is the MDS edit-distance system compensates for this accidental transposition in numbers and will return the records that are similar to what was entered. This helps you find the social security number that was intended to be searched on.

Another common question is, "what happens when you search for a member whose address on file is more current than the one that is being used in the search?". The answer is that historical data is stored in the MDS database in order to keep track of the information that represents a particular member. This is because outdated information is still pertinent to a member's record, regardless of the fact that it is older information. It still represents who that member is. When updating a particular member's attribute, MDS marks the older information as inactive and the newer information as active. This way, when you search for a John Smith that used to live at a particular address, or that used to have a particular telephone number, you will still be able to get him back in your search results. The engine will perform a search on the current and all historical member data to help identify a record. Most search tools do not have this feature.

When asked "what happens when you search for a member whose information is found at a different location, or source, to the one you are searching on?". The answer is that the MDS indexes the entire member base and tracks each member by way of Enterprise ID technology. If a particular member is only found at site A's database, this information can be pulled from it and create a new record that will now reside at site B's database too. The exact same member attributes and data is copied over in order to have the data consistent and up to date. This helps avoid any possible data entry errors that may occur when trying to manually copy information that was only found at one site.

Ultimately, the MDS purpose is to minimize the duplication of data that is primarily caused by human error.



Your focus will now be shifted to a discussion on how data preprocessing is actually a 3-step process.

For a deeper understanding of the material that is covered in this presentation, you should review other IBM Educational Assistant presentations that discuss bucketing and algorithms. However, these are not required in order to understand the material covered in this presentation. Examples from previous slides are reused to discuss data processing.

There are several configurable processes that take place before the insertion of data into the MDS database. This is done in order to create metadata, which allows the engine to perform faster searches when querying the database.

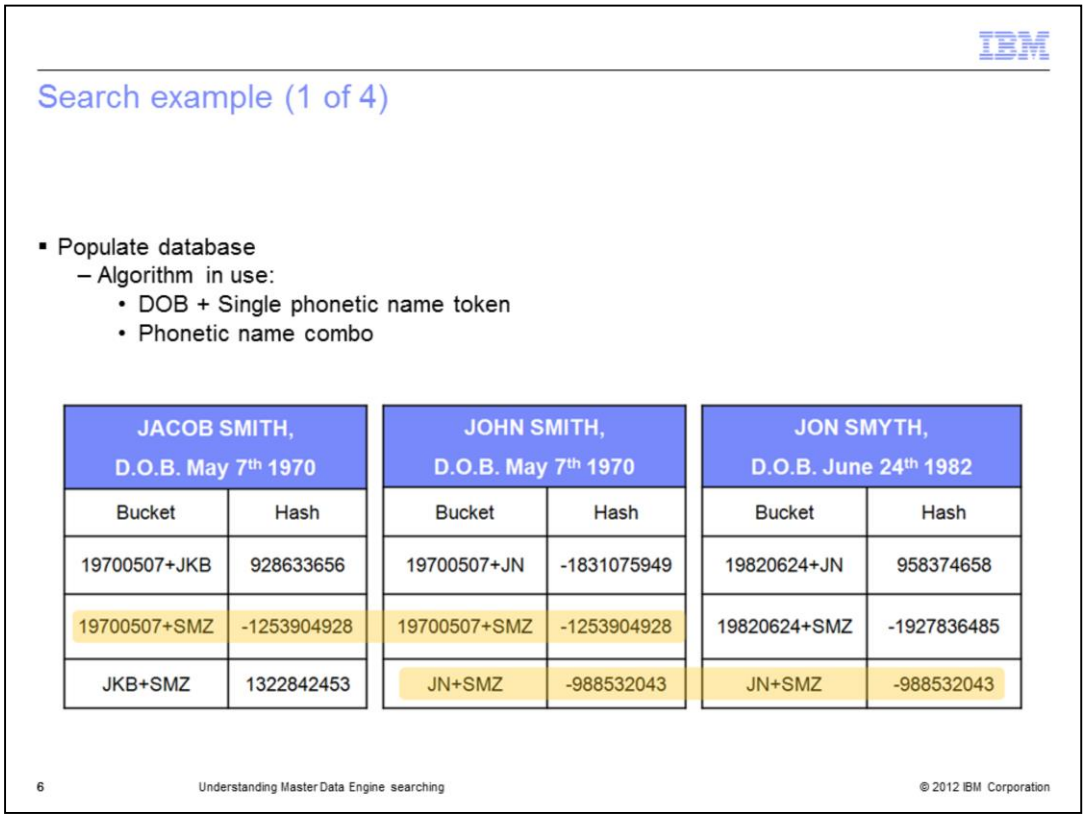
The first step is "standardization". In this step, data is converted into a more consistent, usable format that will make all of the data that comes in uniform. For example; the case of all letters are changed to upper case and special characters like the dashes are stripped out. White spaces are also eliminated as illustrated in these examples.

The second step is "bucketing". In this step, the already standardized data may, or may not, be combined, and is then transformed, or translated, into a numeric hash value. Looking up an integer or hash value yields optimal retrieval of such data from the database. Using "John Smith" as an example illustrates this process. This name search employs the phonetic algorithm, which is discussed on the next slide. Performing the search, yields the phonetic strings of "JN" and "SMZ". They are then combined into one bucket using a secure hash algorithm to produce the negative hash integer value. Since numbers work much faster in the database than any other data type, this hash value is the number that corresponds to this particular bucket, and is inserted into a special table used for quick lookups. You may already be familiar with the mpi_membkd table.

The third step is "comparison". This is the process of comparing standardized data and assigning a score that represents how similar the sets of data are. When performing a search, you provide specific criteria, such as name, telephone number, address, social security number, zip code, and so on. From this search data, buckets are created on-the-fly, and the database is searched for all matching buckets. This is often referred to as "casting a net". A set of members whose bucket values match are returned. These members are called the "candidates". All of the attributes from each candidate are compared with the search data.

Here is where the scoring occurs and is based on individual scores given to distinct attributes of each member. There are different agreement types and disagreement types that contribute to the score as well. Take for example, the exact agreement match of "JOHN" J O H N and "JOHN" J O H N. Or the phonetic agreement match of J O H N and J O N. There is also a nickname match, like "BILL" and "WILLIAM". There is an initial match, like "G" and "GREG", this case will give a positive score because the correct initial letter was entered, or found in the database. Another contributor is an edit distance match, like "FREDERICK" and "GREDRICK", where the edit distance is only one character, the first letter. There are also the disagreement scores, like "JOHN" and "DAVE", where the phonetics and spelling of these names do not match at all, giving a negative score.

Once all of the scores are completed, a final comparison score is given to every member that was returned in the search. It is very typical that many members or candidates are brought back as part of the bucket sharing process. However, those that match closer to the search data have a higher score. The final comparison scores describe how closely your search criteria matches with the retrieved candidates' attributes found in the database.



In this search example, you will gain a better understanding on exactly how searches take place in the database. Some important points of interest include, “What is being done with the data?”. “How is the data injected into the database?”. “What actually happens when you perform a search?”.

This example is composed of a very simple algorithm. It includes the combination of date of birth and a single phonetic name token. The algorithm will also use a phonetic name combination.

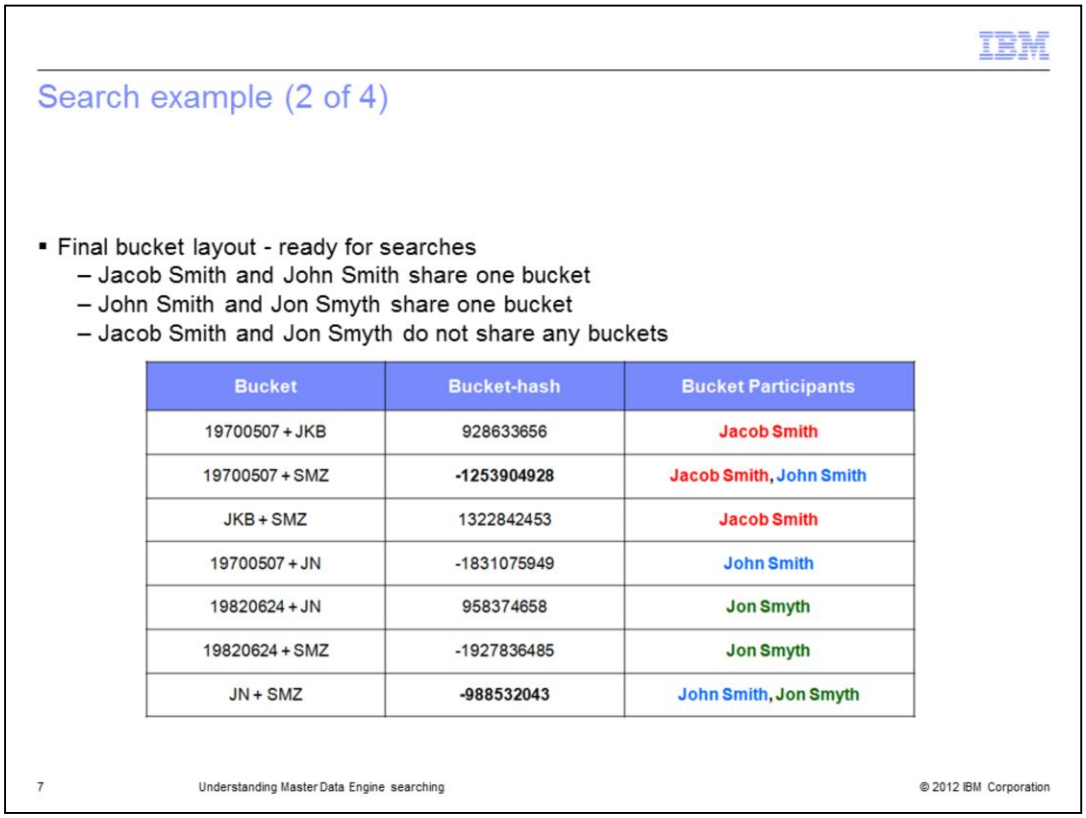
First, the data is populated with three sample members. Each contain a first and last name, and a date of birth.

The first sample member has the name of “JACOB SMITH” with his corresponding date of birth. This creates three buckets, each with their corresponding hash values.

The first bucket is the combination of Jacob’s date of birth and his first name. The second bucket is a combination of his date of birth and his last name. The third bucket is the name combination “JACOB SMITH” which has been converted to “JKB + SMZ.”

The second sample member is “JOHN SMITH” with the same date of birth as “JACOB”. Now you are going to create three buckets with the same standards. The first bucket is the combination of date of birth and the first name using “JN”. The second bucket is the combination of date of birth and his last name using “SMZ”. The third bucket is the name combination “JOHN SMITH” which has been converted to “JN + SMZ.” Notice the hash value is a unique value generated based on the bucket contents, and are used by the system for seeded look-ups.

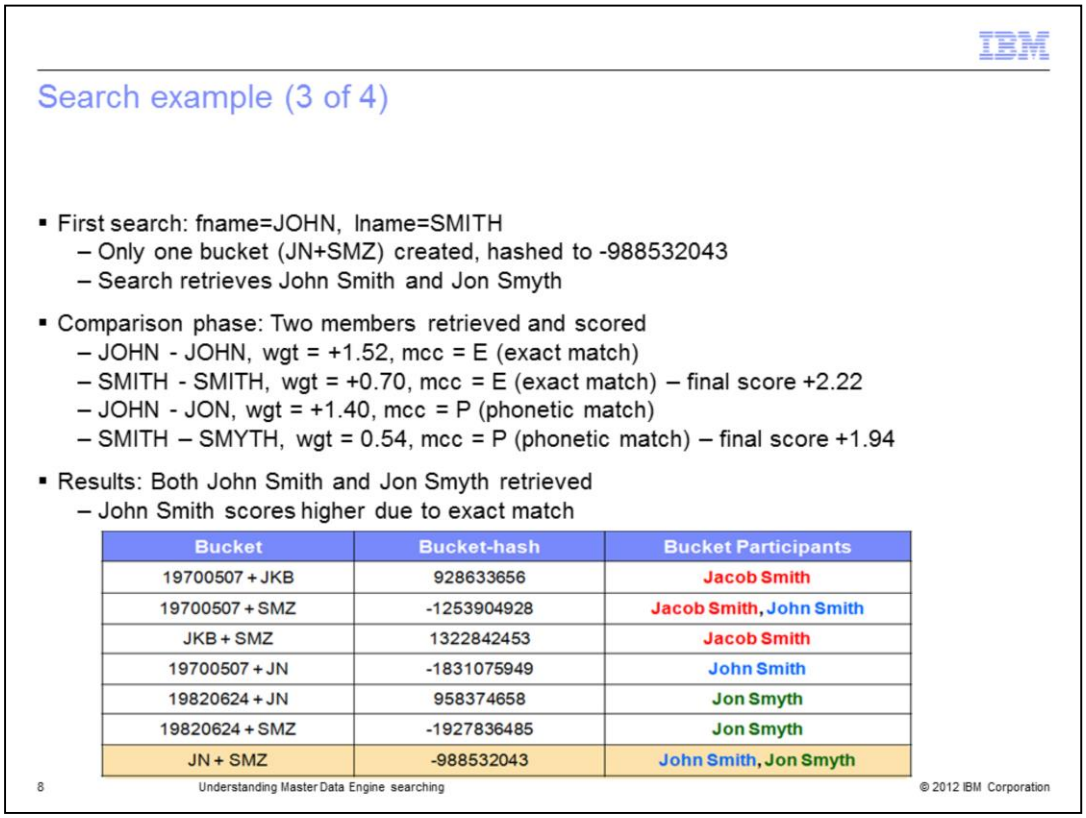
The third name example is “JON SMYTH”, spelled J O N S M Y T H which sounds just like the second member “JOHN SMITH”, spelled J O H N S M I T H but has a completely different date of birth. Again, you use the same standard to generate its corresponding buckets and hash values. If you compare all three members, you notice that there are some overlaps or the sharing of buckets.



The results from the previous slide are displayed on this slide in the table. You can see all buckets, relevant bucket-hashes, and the records that are participating in each bucket.

The first bucket pair is between “Jacob Smith” and “John Smith”. They have the same date of birth and the same phonetic last name “SMZ”.

Next, “John Smith”, spelled J O H N S M I T H and “Jon Smyth” spelled J O N S M Y T H are sharing a bucket with their phonetic name combination of “JN+SMZ”, because their first and last names sound alike. Finally, notice that “Jacob Smith” and “Jon Smyth” do not share any buckets.



Now that the data is in the Hub, a search is performed that helps you better understand how the bucket retrieval and scoring mechanism work.

The first scenario is an example of a search using only a name as the search criteria. You typed the first name “JOHN” and last name “SMITH”. Notice that the date of birth was not used.

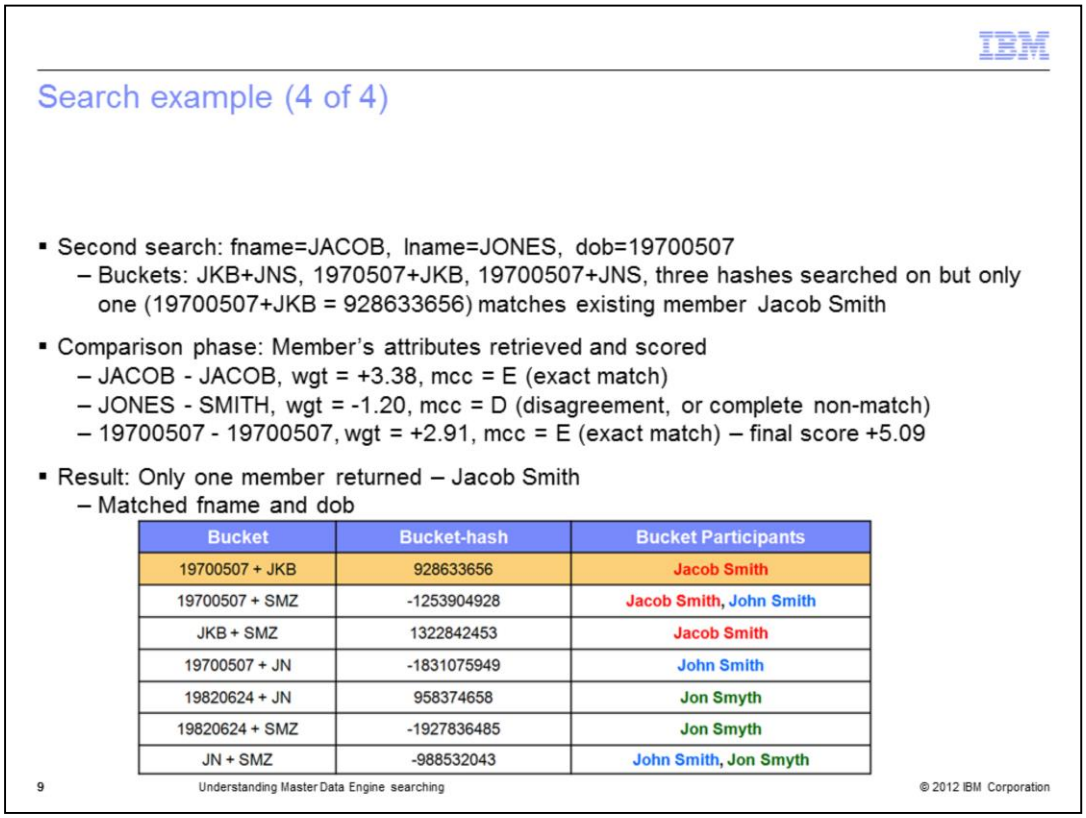
The algorithm has a phonetic first name and last name bucket combination. Therefore, the query only generates one bucket: “JN + SMZ”. The system then hashes this bucket as -988532043 and retrieves all the records that share this particular value. This search produces “John Smith”, J o h n S mi t h, and “Jon Smyth”, J o n S m y t h.

The system then goes through the comparison phase and compares the comparison strings for the retrieved records with those that were provided in the search criteria. These are scored to determine how similar the results are to the actual input that was provided as the search criteria.

The comparison yields both “J O H N” and “S M I T H” as an exact match. Therefore, the final score for this particular member is a +2.22.

Next, you have “J O H N” versus “J O N” and “S M I T H” versus “S M Y T H”. Both sets match phonetically. The final score for this comparison is a +1.94. Even though this candidate did phonetically match, it did not score as high as the previous candidate.

Since the first candidate, “J O H N S M I T H” scored higher due to an exact match, you can say that this member is the one that most closely matches the search criteria.



This scenario is similar to the previous one, but with a slight difference.

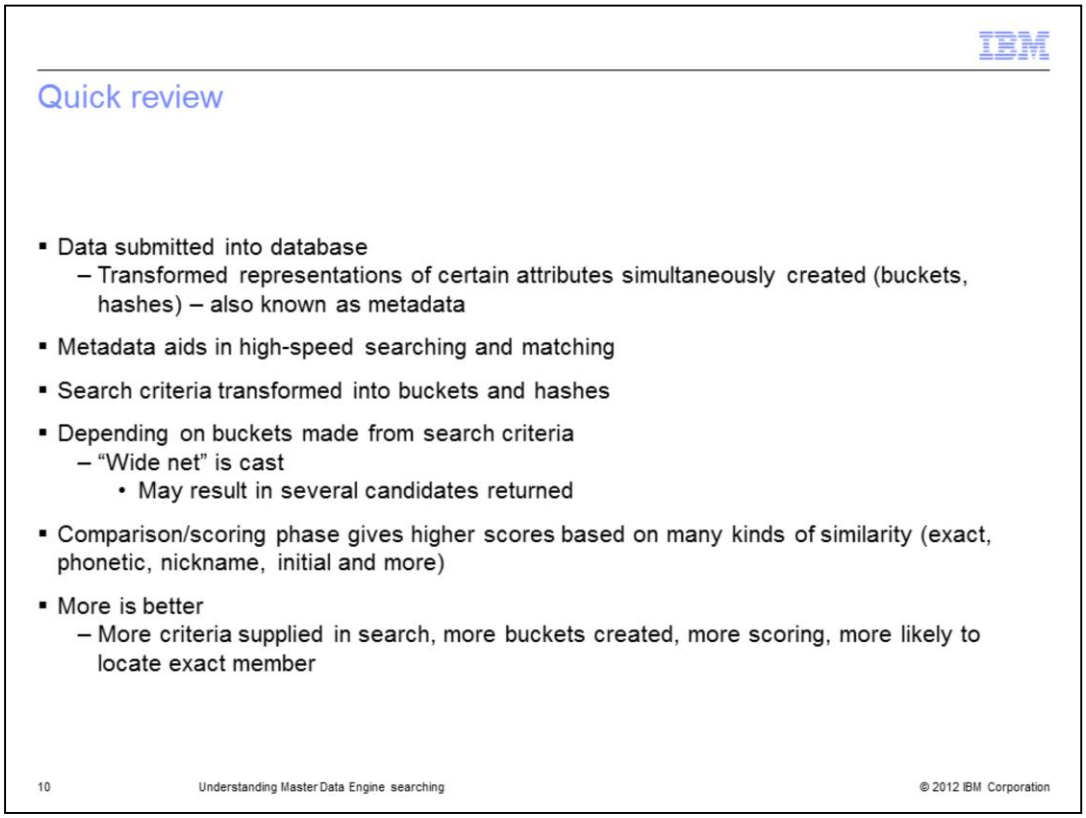
You search for “JACOB JONES” and this time you also enter a date of birth “1970-05-07”. This creates three buckets. One with the combination of the first and last name of “JKB+JNS”, a second bucket with the date of birth and the first name as “19700507+JKB”, and a third bucket with the combination of the date of birth and the last name as “19700507+JNS”. Each of these generate their own, different hash value.

The engine searches for these newly generated hash values, and find that only one existing bucket, “19700507+JKB = 928633656” matched your search criteria. This is the date of birth plus first name bucket that has the existing record “Jacob Smith”. Recall that there is no “JONES” in the system, so no buckets exist that contain the name “JONES”.

Since the search retrieved one member, “JACOB SMITH”, the system now has to score its data through the comparison phase. Again, utilizing the string yielded by the input search data.

The input comparison string for “JACOB JONES” is compared against the database record’s comparison string for “JACOB SMITH”. “JACOB” versus “JACOB” is an exact match, yielding a positive score. “JONES” versus “SMITH” is a total disagreement. The names are not close at all, hence, the last name comparison receives a negative score. Finally, the date of birth is an exact match and therefore receives a positive score.

After adding up all of the scores, you can see that the final score stays at a positive +5.09, despite the last name being in disagreement. It is possible that the last name of this record used to be “JONES”, so the MDS score helps find the best possible match through its scoring mechanism. In this case, you can see that having an exact match on the first name and the date of birth helped drastically. Therefore, based on your input search data the most likely candidate is narrowed down to “JACOB SMITH.”



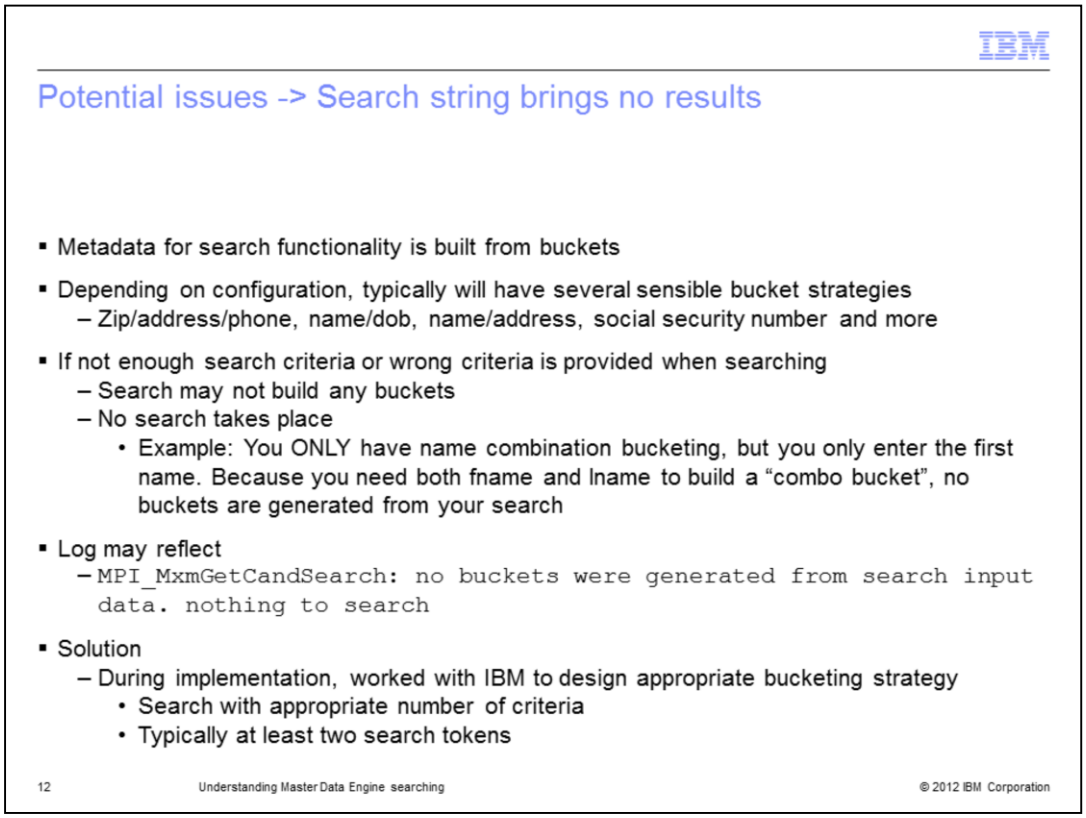
It is time to recap some of the information that you have learned so far. The data that comes into the MDS system receives pre-treatment that helps in the creation of buckets with associated hash values. Certain attributes are transformed in order to create representations of this data. This is what is referred to as the metadata.

It is this metadata that is used in order to perform high-speed searches and achieve the results that the software can deliver. When searching, the bits of information entered in the search criteria are also used to create temporary buckets and hashes. These are then used to look up any existing matches of this same metadata in the database.

When several buckets are generated due to the amount of information provided in the search, a "wide net" is cast which may result in the retrieval of many potential candidates. One may be miss led by this, thinking: "Well, if too many results are returned, then it defeats the purpose of the search.". It is common for dozens of candidates to be returned, nonetheless, the high-speed comparison phase will help score each of the returned members and list them according to their probability of being the actual record of interest. Results are returned in the order of most-likely to least-likely candidate.

Finally, it is important to emphasize the fact that "more is better". To achieve best results, you should enter as much information as you can. This allows you to retrieve as many members as possible to participate in the comparison process. This method is called "casting a wide net". This gives you a higher chance of locating the member that you are interested in finding, quickly and effectively.

The next slides cover some of the most common issues that you might encounter when searching for records using the MDS search engine.



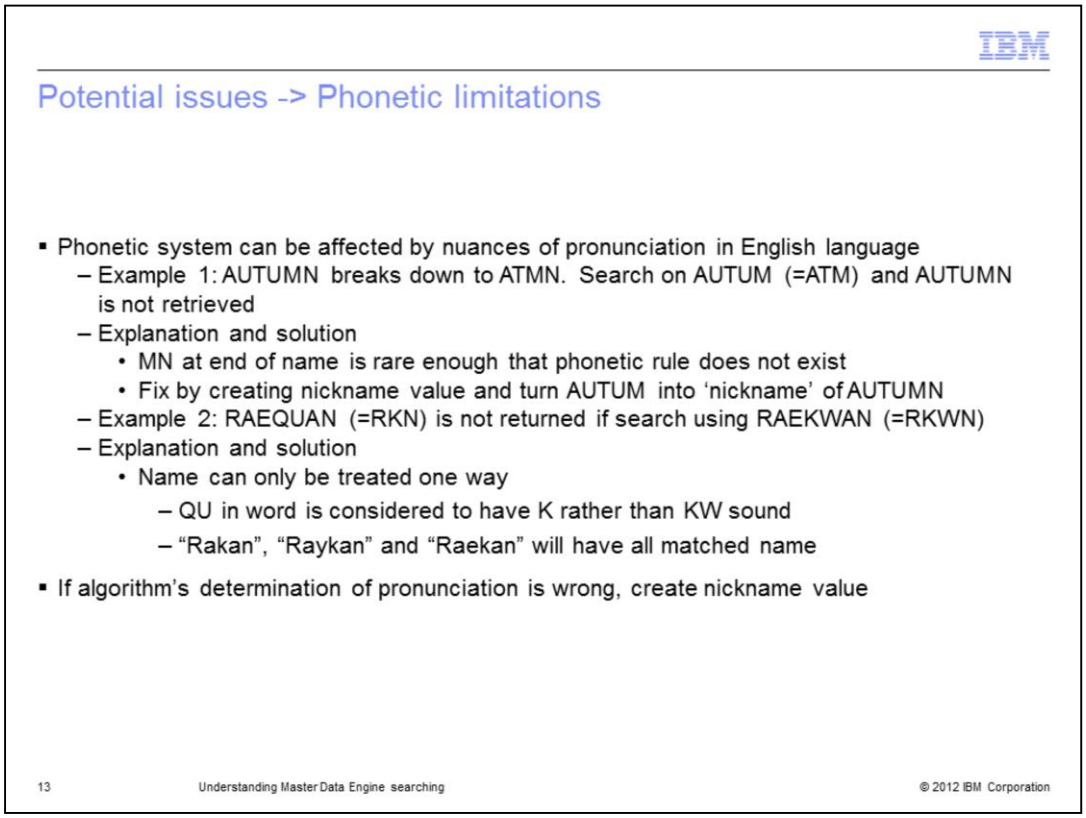
The first issue is when searching brings back no results. As discussed earlier, the metadata that gets created is solely dependent on the number of buckets that are created. This needs to be strategically configured in order to enable the creation of the most relevant attribute and data combinations that are found in the buckets. Combinations like name plus address, name plus date of birth, social security number only, and so on, can help improve the way that data is put into buckets and improve the searches. This is important to consider. For instance, suppose you wanted to create a bucket that takes the combination of last name and state. As you can already anticipate, this bucket will turn out fairly useless as way too many members can belong in it. Just imagine having a listing of all of the Johnsons that live in a particular state. Simply put, this is not a very effective bucket. This is why strategic buckets need to be set in the algorithm during your implementation phase.

If you do not enter sufficient information that is relevant to the bucketing algorithm, you will not have any buckets to use as reference to search the database. For example, you are not able to build and retrieve any buckets, and no search can actually take place.

For example, if the bucketing algorithm requires that you enter a first name and last name in order to create a bucket, and if you only enter a first name in your search, then no buckets can be generated. This is because the algorithm requires both first and last name in order to have something to search against the buckets that are already stored in the database.

This kind of mistake will cause an error to appear in your log files, like the one displayed on this slide. This error indicates that the search that you have just made did not generate enough buckets according to your algorithm.

One solution to this issue is to ensure that the appropriate number of search criteria is entered. Typically, this requires that you enter at least two search tokens. So if you search by only entering one name token, chances are that no buckets will get generated solely from the one piece of information. No results are returned by the MDS.



Another issue is phonetic limitations. The phonetic system can be affected by nuances of pronunciation in the English language. Next are a couple of real world examples that some customers have actually run into in the past.

The first example is when you are searching for AUTUMN, spelled A U T U M N. This name actually breaks down as A T M N. If you search for A U T U M instead, which sounds like AUTUMN because of its ending silent letter, the breakdown is actually A T M. Since the ending M N is rare enough, a phonetic rule does not really exist for it. So AUTUMN, spelled A U T U M N, is not retrieved. You can fix this problem by creating a nickname match value using the IBM Initiate Workbench. Make AUTUMN, spelled A U T U M without the N, a nickname for AUTUMN with an N. This way when you search for A U T U M, a nickname match for AUTUMN will also be retrieved and you are able to find your match.

The next example is with the unusual name of "RAEQUAN". This one phonetically breaks down to R K N. If you search for R A E K W A N, it will phonetically break down to R K W N and will not bring back the first name. To come up with an explanation for this, you must first understand how the name is actually pronounced. You probably pronounce it as RAY-KWAN. However, the system can only treat this name in one way and when it sees the Q U in the name, the phonetic algorithm treats it as a K, rather than a K W sound. This is the rule of the Q U in the middle of a word. "R a k a n", "R a y k a n" and "R a e k a n" will all match this name perfectly because they have been broken down to R K N, also.

But since you did not type in phonetically the way that the system believed it should be pronounced phonetically, you cannot get the first name back. These are really rare instances in which the problem can be corrected by creating a nickname value for them. This brings back the member in question, despite the fact that you may not type it in correctly. The phonetic algorithms can make mistakes as they are not 100% perfect, so the system might get names wrong sometimes. Nevertheless, it can be corrected easily.

Additional resources

- Other related IBM Educational Assistant presentations
  - Introduction to algorithms
  - Introduction to buckets
- Documentation
  - Algorithm Manager or Workbench User Guide - http://publib.boulder.ibm.com/infocenter/initiate/legacy/index.jsp

This slide displays a list of additional resources available to you that contain more information regarding the topic of this discussion.

Trademarks, disclaimer, and copyright information

IBM, the IBM logo, ibm.com, Initiate, and Initiate Master Data Service are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of other IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM'S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS OR SOFTWARE.