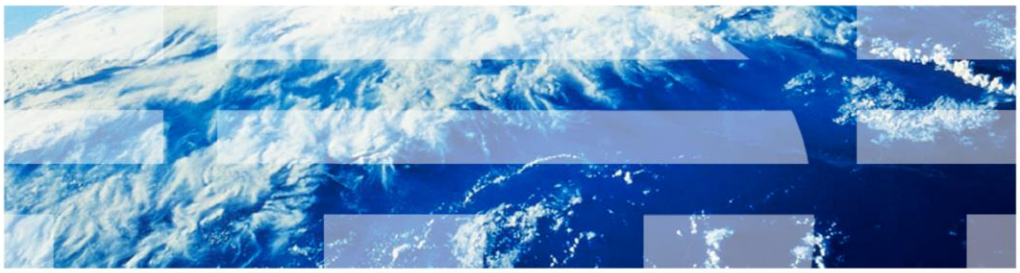
IBM

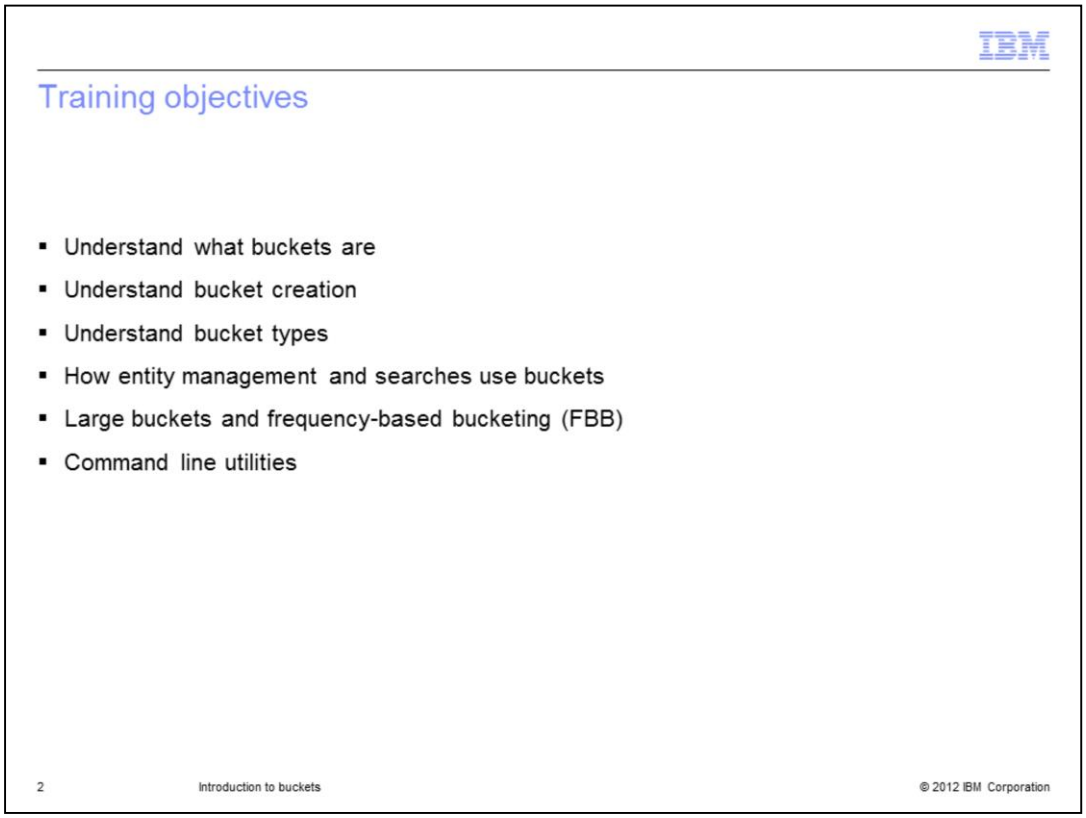
# IBM Initiate

## Introduction to buckets





This is the IBM Initiate® “Introduction to buckets” training presentation.




The objective of this presentation is to introduce you to how the IBM Initiate Master Data Service®, referred to as MDS, uses buckets for searches and comparisons. This presentation begins by discussing what buckets are and how they are created. By the end of this presentation you should understand the different bucket types and how entity management and searches use buckets. You will learn about large buckets and a concept referred to as “frequency-based bucketing” and finally, you will discover the purpose of some useful command line utilities.

## What are buckets?

- Group of members or data numerically or phonetically similar and share common attributes
- Bucket data is output of derivation attributes configured to participate in candidate selection

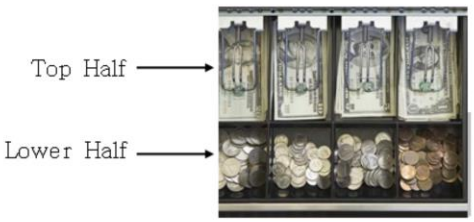



Before you can learn how buckets work, it is important to understand what they are. Buckets are defined as a group of records that are numerically or phonetically similar and share common attributes. Bucket data is the output of the derivation attributes configured to participate in the candidate selection process.

Essentially, buckets represent the organization or grouping of data that is similar.

## Bucket analogy (1 of 2)

- Analogy
  - Bucket data represented by bills and coins
  - Cash register's drawer composed of two divided storage areas
    - Top half for bills, lower half for coins
  - When giving change, go to bottom storage area where coins have been pre-determined to be located




4 Introduction to buckets 

To understand what buckets are, a simple analogy is used. In every grocery store you visit you will typically see cash registers. The cash registers contain a drawer that is typically divided into two sections; the top half for bills and the lower half for coins.

When the cashier gives you change for a dollar, they do not go to the top half of the cash drawer. Instead, they go directly to the bottom half where they know the coins are kept. Imagine how much slower each cash transaction will be if the cashier has to use a single cash "jar" instead of a cash register to give the customer change.

## Bucket analogy (2 of 2)

- Use buckets to eliminate unnecessary processing of data
- Increases efficiency by going directly to data that is known to be similar

| | |
|---|---|
| **Name:** | Jonathan Green |
| **Address:** | 2717 111th Avenue |
| | San Francisco, CA 94115 |
| **Gender:** | Male |
| **MRN:** | IH:987754 |
| **SSN:** | 888-22-6446 |
| **DOB:** | 1970-05-08 |

| | |
|---|---|
| **Name:** | Patty Countryman |
| **Address:** | P.O. Box 2919 |
| | Concord, CA 94520 |
| **Gender:** | Female |
| **MRN:** | RMC:870504 |
| **SSN:** | 263-19-9065 |
| **DOB:** | 1965-08-02 |

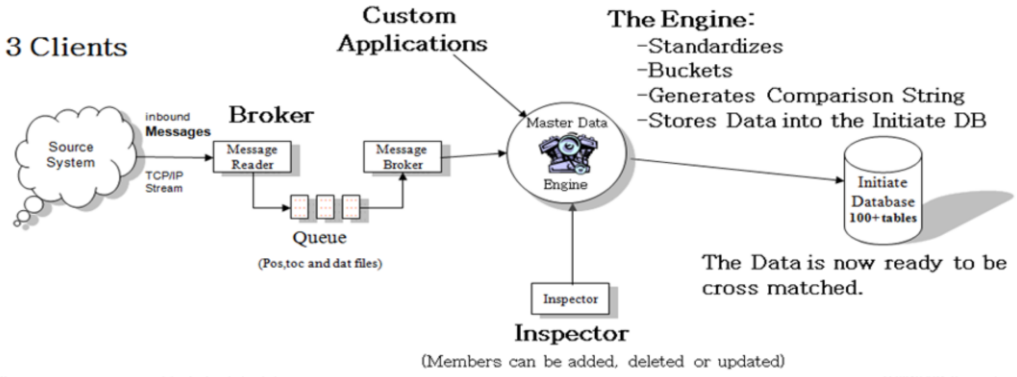
5 Introduction to buckets 

Buckets work in much the same way. Just as you would not search for a penny in the bills bucket, you would not want to spend time comparing two sets of member's data that are completely different. For example, the two members displayed on this slide would have their data stored in completely separate buckets.

Conceptually, buckets work the same way. By using buckets, you eliminate unnecessary processing of data that is completely different. This exponentially increases efficiency by going directly to the data that is known to be similar. For example, take the two members displayed on this slide. These members have completely different names, addresses, gender, and so on. Due to their data disparity, they will never generate common buckets. Therefore, searches for any attributes associated with one of the members will not waste any time comparing attributes associated with the other. It is this organization of attributes or buckets that makes the Master Data Service so efficient.

## Bucket generation triggers

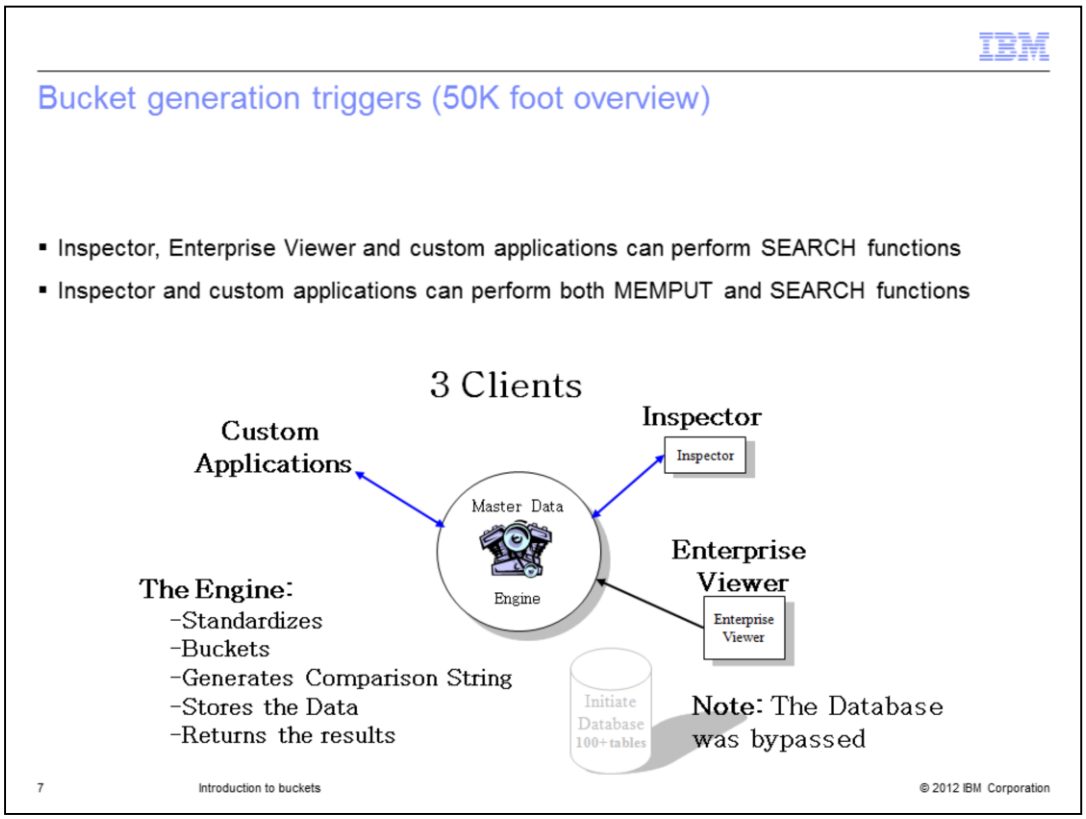
- Creation of buckets are initialized or triggered through **MEMPUT** or SEARCH interactions
- Three example clients that generate **MEMPUTs**
  - Custom applications
  - IBM Initiate Inbound Broker
  - IBM Initiate Inspector





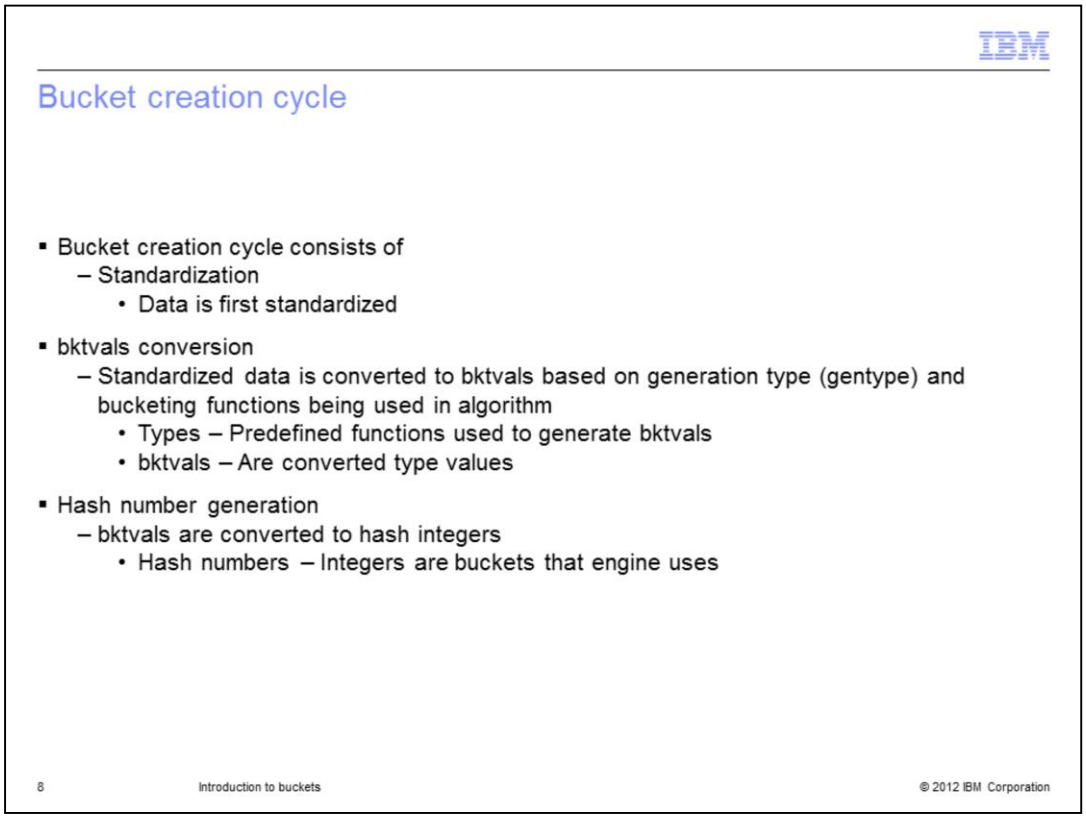


This slide illustrates how buckets are generated. The interactions that trigger bucket generation are called member puts, referred to as MEMPUTs or searches. On this slide, there are three example clients that generate MEMPUTs: Custom applications, the IBM Initiate Inbound Broker, and IBM Initiate Inspector. When a MEMPUT call is made by any one of these clients, the Master Data Engine takes the data and standardizes it, creates buckets, generates the comparison string and finally, stores the data into the IBM Initiate database. The data is now queued and is ready to be cross matched against all members that share common buckets.

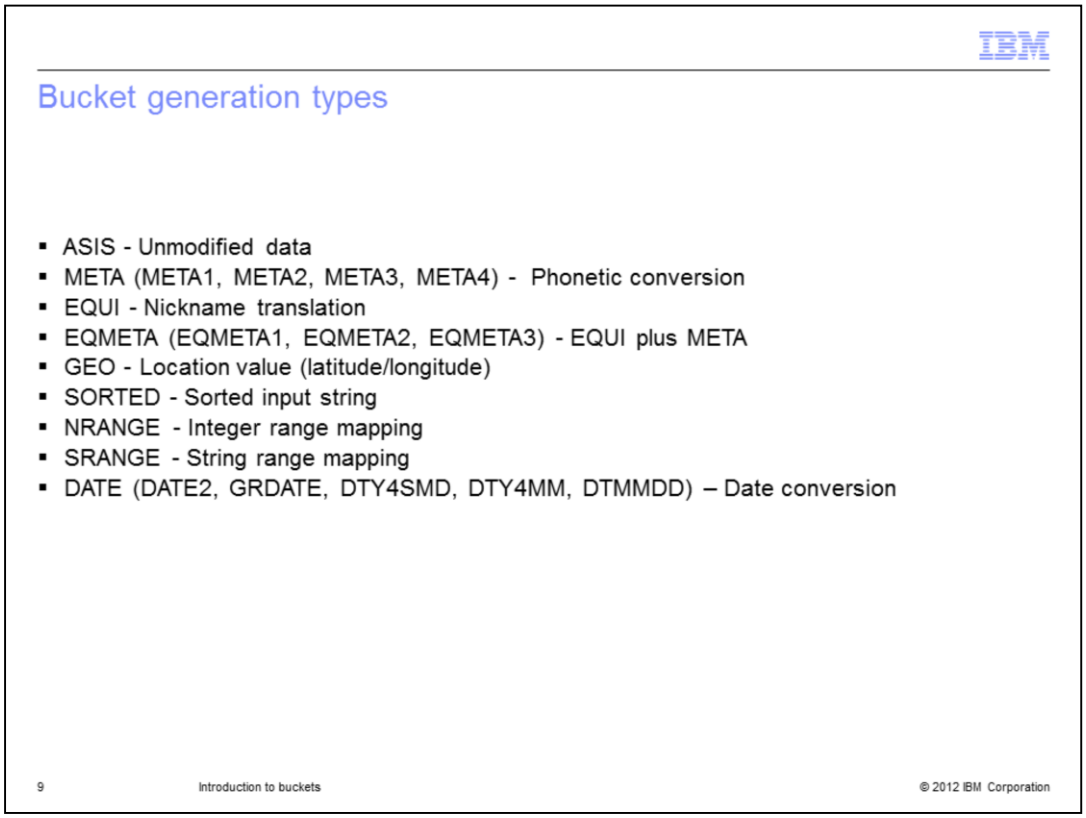



For search interactions, the same three example clients are used. The custom applications and Inspector can perform both the MEMPUTs and search interactions. The front end application, Enterprise Viewer, is only capable of performing searches.

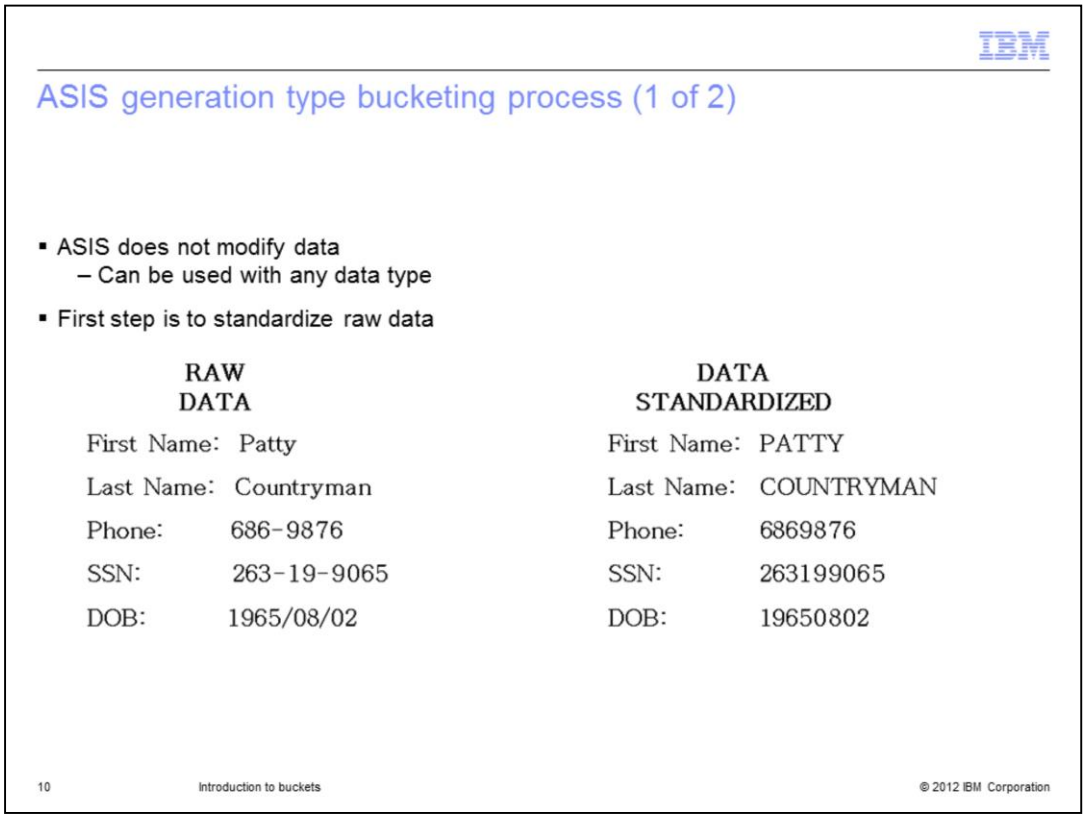
After the search criteria has been entered, the engine standardizes and derives the data, returning the results to the respective clients. The data that is used to retrieve the matching buckets is not stored in the database, in other words it is transient. This temporary information is only used in the comparison process.



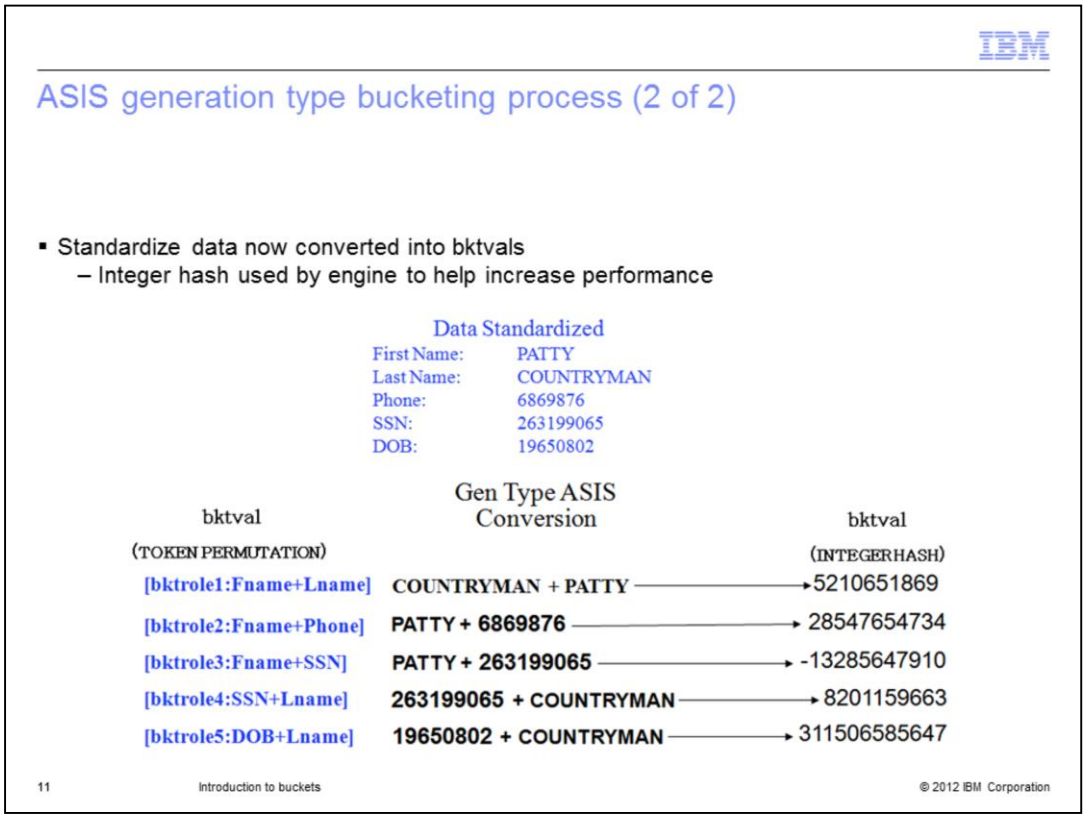

The creation of buckets consists of cycles. Standardization is where the data is converted into caps and all non-print characters are stripped. The system can also be configured to take all street names with the value of “West”, for instance, and format it as “W”, or change all occurrences of “Street” to “St”. The standardization process will also strip name prefixes like Mr., Mrs., and so on. Bucket-val conversion is where the data is converted to the type that is being used. Types are predefined functions used to generate bucket values. The bucket-vals themselves are the converted type values. The final cycle is hash number generation. A hash number is an integer representation of a bucket-val. This is the bucket that the engine actually uses.



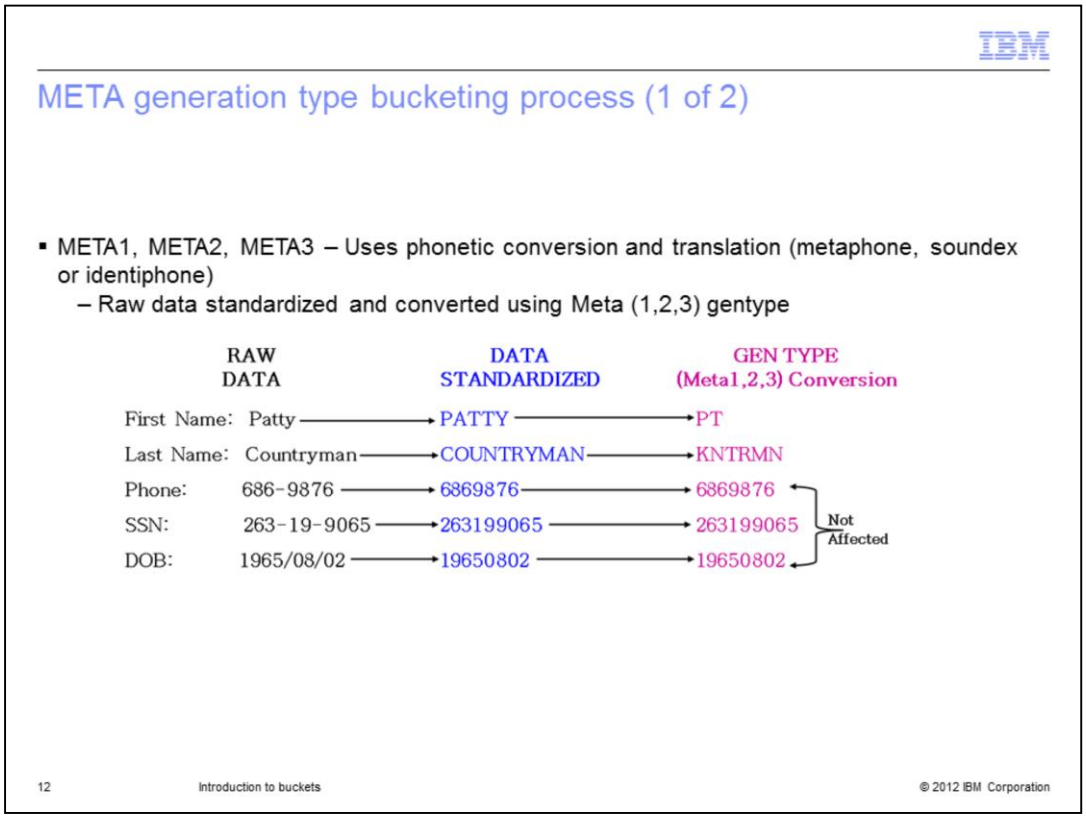

There are several Bucket Generation Types, referred to as gentypes. There is ASIS, META, EQUI, EQMETA, GEO, SORTED, NRANGE, SRANGE, and DATE. This presentation only discusses the top four gentypes.



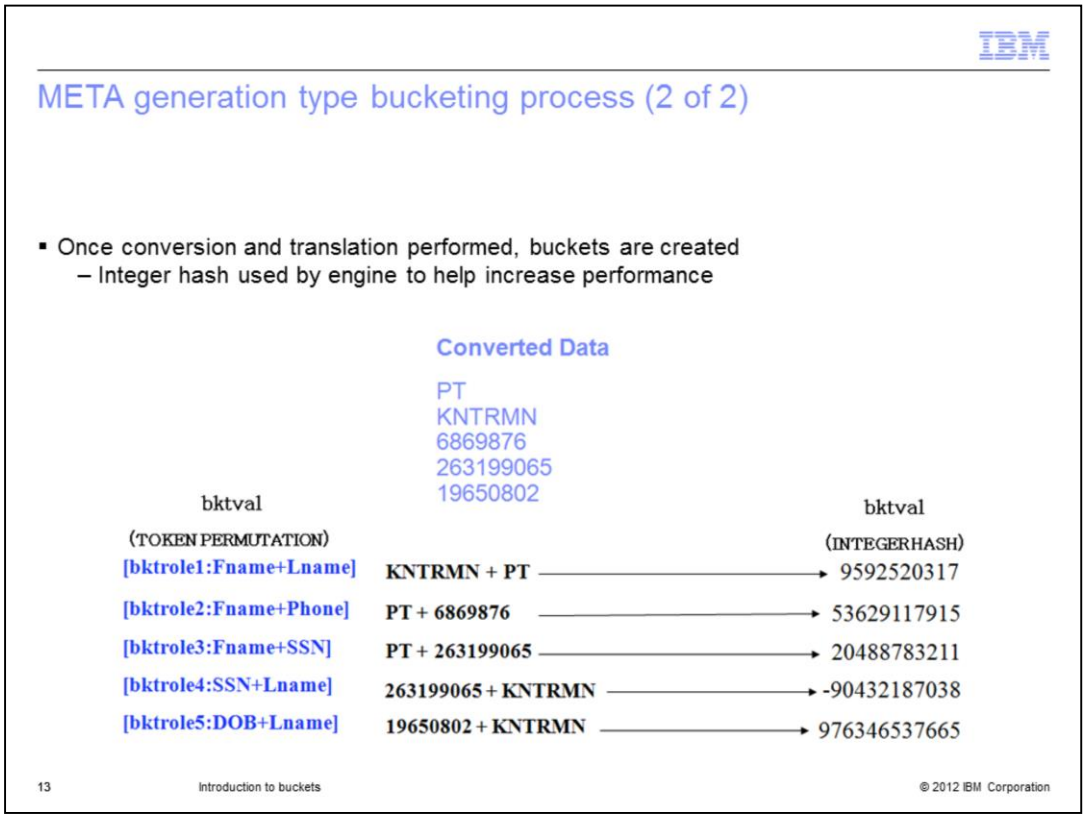

First of the gentypes is “ASIS”. The ASIS gentype does not modify the data after standardization and can be used with any data type. If you take a look at the example provided, the raw data contains the first and last name, telephone number, social security number and the date of birth of the record. In this example, the standardization process will capitalize all the letters and remove any of the non-print characters before the ASIS phase begins.



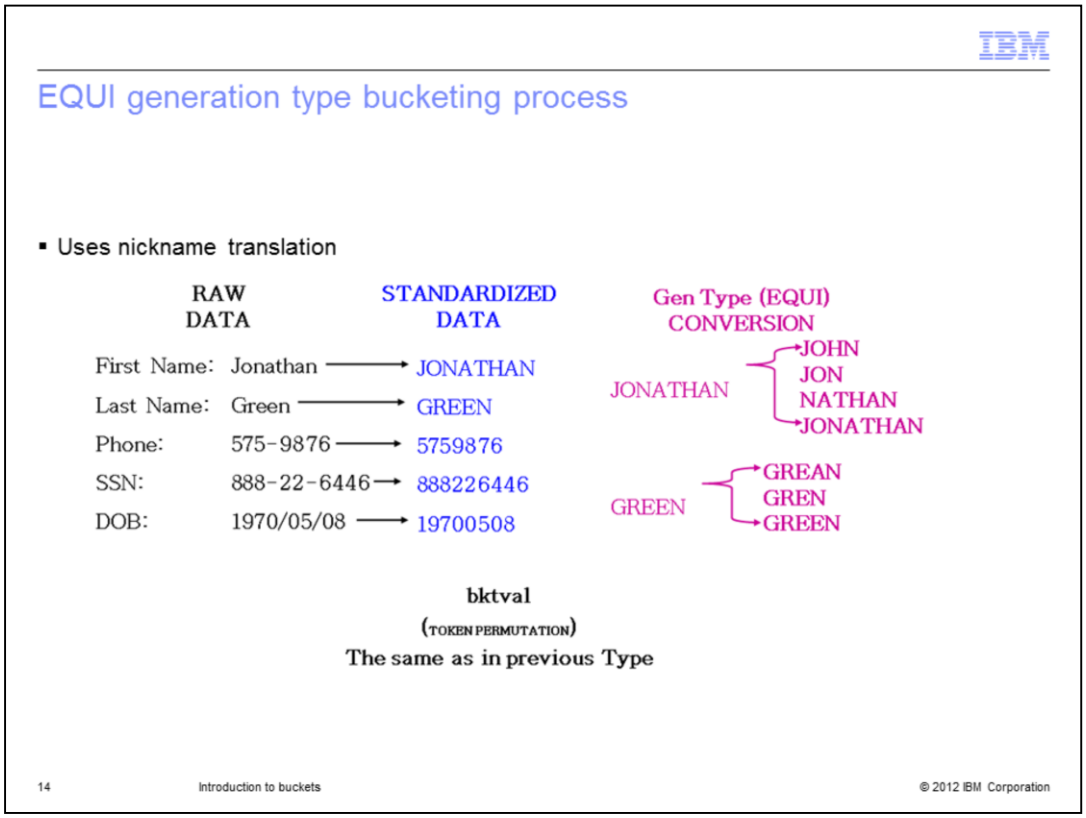

Once the data is standardized, the gentype of "ASIS" is applied. In this case, there are no further transformations to be made, because the information should be used “as is”. The bucket-val or token permutation is then created for bucket role 1 which is a combination of the first and last names. You may notice that the last name is presented first; this is because the bucket-val is always in alphabetical order. This combination is then converted into a bucket hash. Next is bucket role 2: first name and telephone number. This is also converted to a bucket hash. Bucket role 3, first name and social security number, is converted to a hash, then bucket role 4, social security and last name, and finally bucket role 5, date of birth and last name. Remember that the bucket hash is the final integer value that the engine uses. Integers are used to maximize database lookup speed.




Meta1, 2 and 3 are the next bucket gentypes. These use phonetic conversion to break down the alphabetic characters into “sound-alikes” code. The example once again, uses the same data and standardization process as in the “ASIS” example on the previous slide. However, once this initial phase is complete, the data will then be converted using the Meta1, 2, 3 phonetic algorithm, taking the first name of Patty and converting it to PT. The last name of Countryman is converted to KNTRMN. These phonetic codes will result from similar sounding names such as PAT and PETE, and perhaps COUNTERMAN, and serve to compensate for spelling errors. Notice that the numerical values were not modified, since this particular gentype will only convert alphabetic characters.



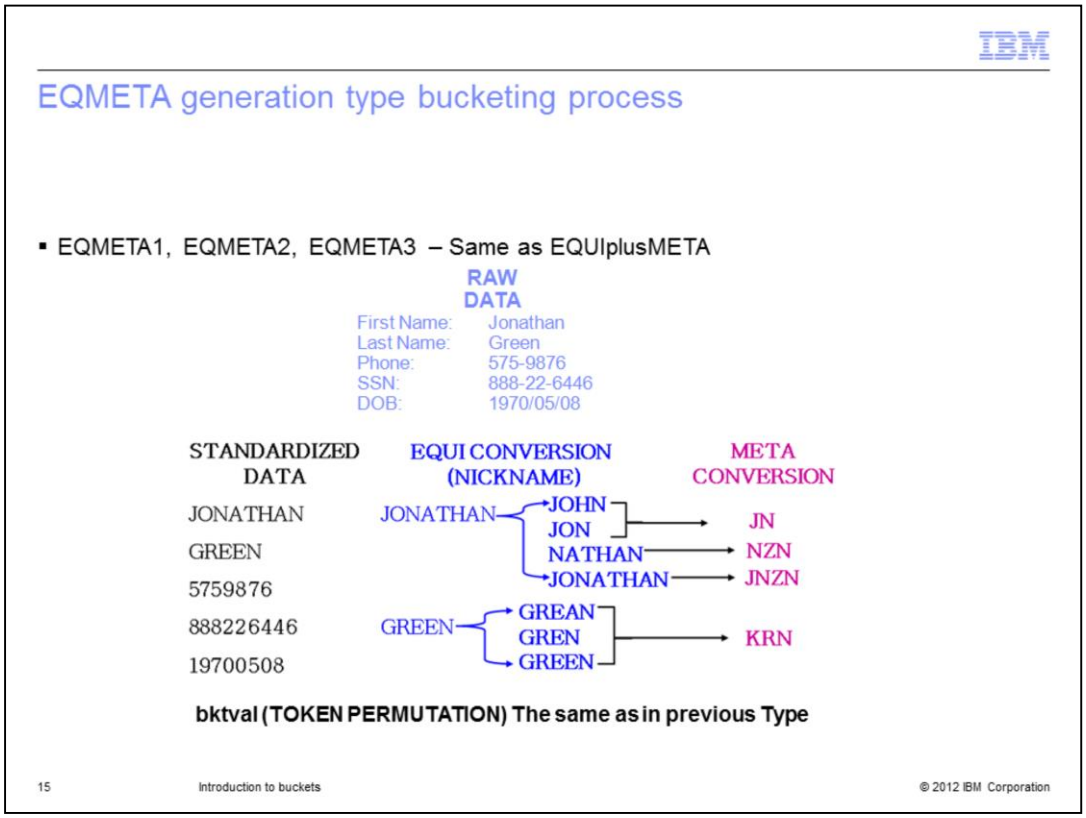

Following the Meta1, 2, 3 transformation, the data is once again converted into a bucket-val permutation. This produces first name plus last name, first name plus telephone number, first name plus social security number, social security number plus last name, and date of birth plus last name. Then, all of those are converted to their respective bucket hashes. All of the values to the right are the buckets the engine will use. If you review the bucket hashes from the previous example, you will notice that these are completely different. The previous bucket hashes were using the gentype of ASIS. In this example the gentype of META1, 2 and 3 is being used.




Now for the EQUI gentype:

This time a different example is employed. In the last examples, the member “Patty Countryman” was used. This time, “Jonathan Green” is the member. The EQUI gentype uses the nickname translation.

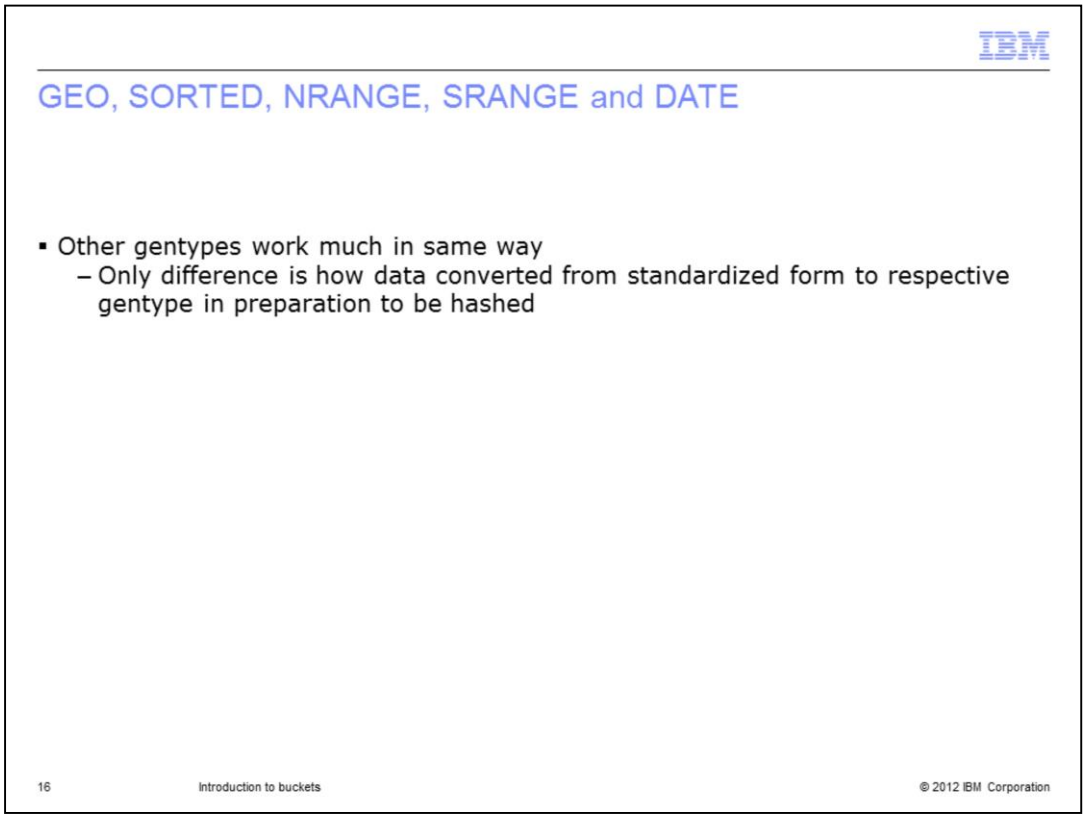
First, the data needs to be standardized. Once this is complete, the data is converted using the nickname translation. The system gathers a list of all the names where “Jonathan” is considered a nickname. Since it is not considered a nickname, only “JONATHAN” is used to generate buckets. However, in the case of a search, if the input data is “John”, the system will then generate search bucket-vals for “JOHN” and “JONATHAN” since “JOHN” is considered to be a nickname of “JONATHAN”. The same applies for the last name. If a search entry has “G-R-E-E-N”, the search will generate bucket-vals for “G-R-E-A-N”, “G-R-E-N” and “G-R-E-E-N”. Typically, last names do not have nicknames, but it is recommended to use common misspellings as nicknames. It is very common for names to be misspelled when entered into a system. By assigning those erroneous names as nicknames you are able to cast a wider net when attempting to locate a record. From this point, the normal path is followed and bucket-vals are created. The difference here is that you have the ability to search on records using the configured nicknames. Searching for J O N G R E N brings back result sets which include JONATHAN GREEN.



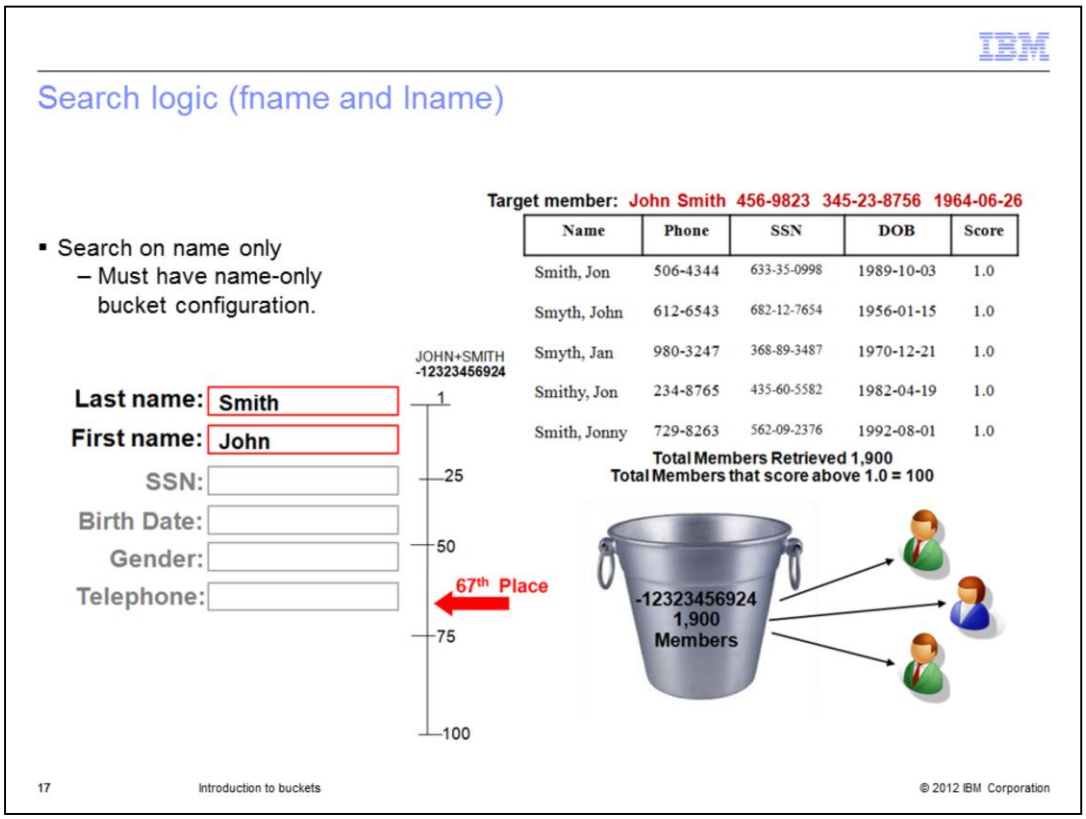

The EQMETA is the combination of EQUI and META. Now, using the same member, the data is standardized, only this time, the gentype EQMETA is applied. This example starts by using the EQUI gentype to generate the nicknames for the first name, again these are John (J O H N), Jon (J O N), NATHAN, and JONATHAN. Next, the META conversion is applied to the first name. In this case, they are JN for J O H N, JN for J O N, NZN for Nathan, and JNZN for Jonathan. Since JN appears twice, only one is used, reducing your first name count.

The same thing is done for last name. First, the list of nicknames is created. Then, the nicknames are converted using META. Since the phonetic conversion of all the last names are the same, they are condensed into a single value of “K-R-N”. Since the last name was short and monosyllabic, the number of permutations is significantly less.

Just as before, the normal path is followed from here: the bucket-vals are created but this time, only three first names and one last name is used.



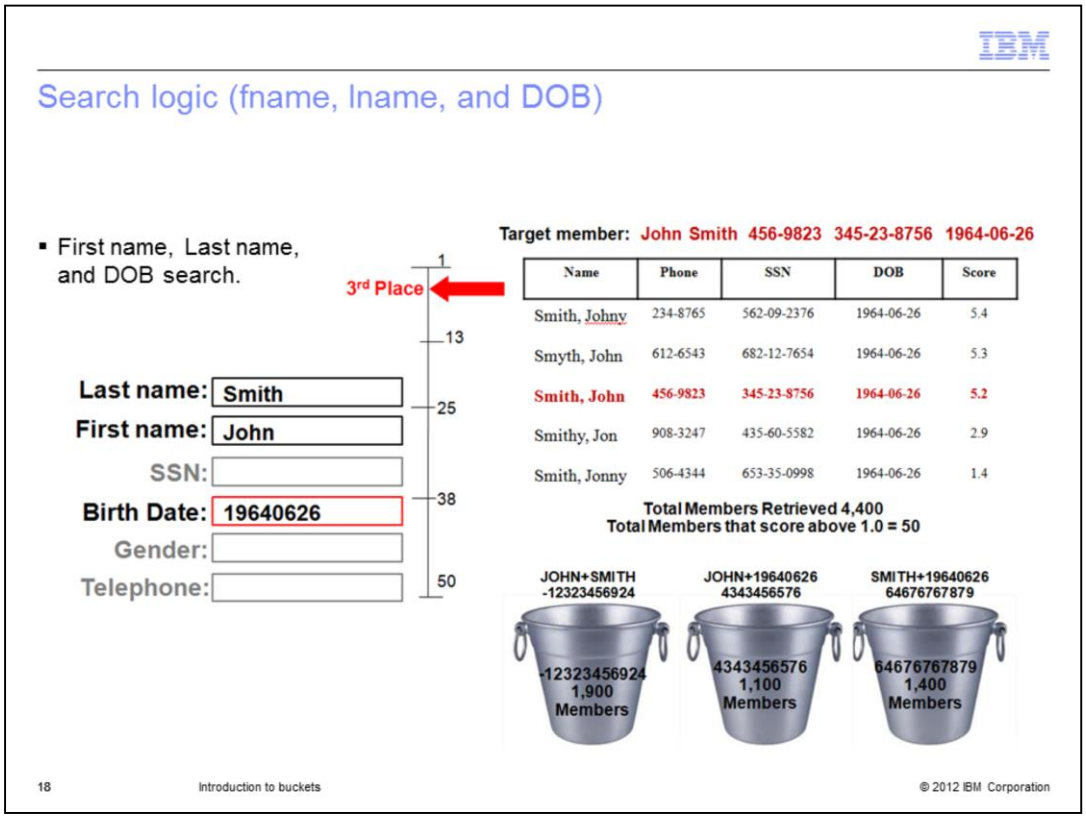

With regards to the standardization and the hashing of the bucket-vals, the remainder of the gentypes work in the same manner. The exception, however, is that the data is converted into a different bucket-val format.




This example takes a closer look at how the search logic uses buckets. Suppose you are looking for John Smith. You have the member's first and last name, their date of birth, social security number and telephone number. Instead of entering all of the information, you decide to only use the first and last name to look for the member. By doing this, the bucket of John plus Smith is created. This bucket is then extracted from the system. The total number of members retrieved is 1,900. The system is using the bucket hash for increased speed of lookups.

Out of the 1,900 rows, only 100 score above 1.0. Out of these 100 rows your search member scores 67th place. This means that you need to review 66 other records before you find the one record you are searching for. Since all of these members score the same, their order shown is somewhat random and is displayed based on the order in which they were retrieved from the database.

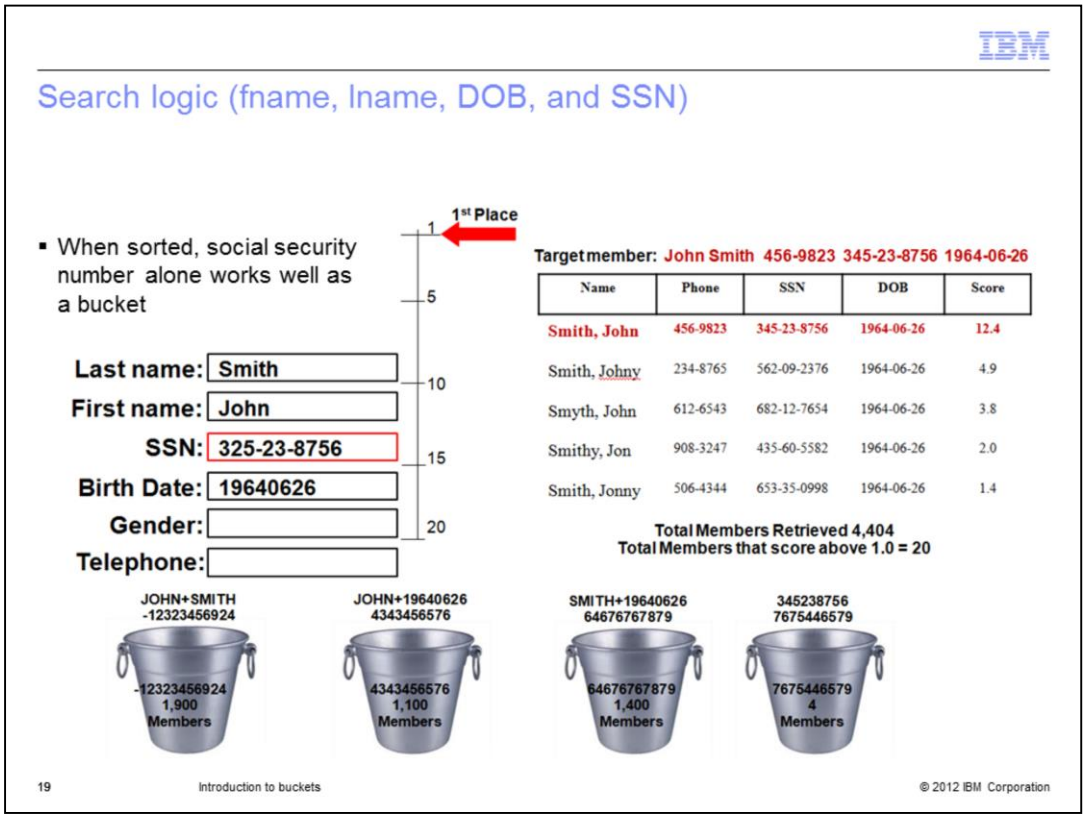
Keep in mind that your system must be configured to use a first and last name search at the very least. Depending on your unique setup you may not be able to use a first name, last name search only. In some configurations, your system will require additional information to be entered.




Now you can see what happens when you use the first and last name and the date of birth to search for the member. In this case, you are going to get the bucket you had before, plus two additional buckets. You will get the John plus Smith bucket, the John plus date of birth bucket, and Smith plus date of birth bucket. Just like before, you will retrieve all of the members. This time you have retrieved 4,400 members. The number of members that scored over 1.0 is 50.

Before you had 1,900 members and 100 members scored above one, this time you have half as many that score above one. The reason for this, is due to the addition of an attribute to your search criteria. This caused 50 of the members that previously scored above a 1.0 to score much lower this time since their date of birth did not match the date of birth you provided in your search. Since their date of birth is different, a negative score can now be applied to the overall score bringing them to below 1.0.

Just by adding the additional attribute of date of birth, your member has risen from 67th place to third place. This makes finding your member much easier. Negative and positive scores increase the likelihood of a search returning the correct candidate.

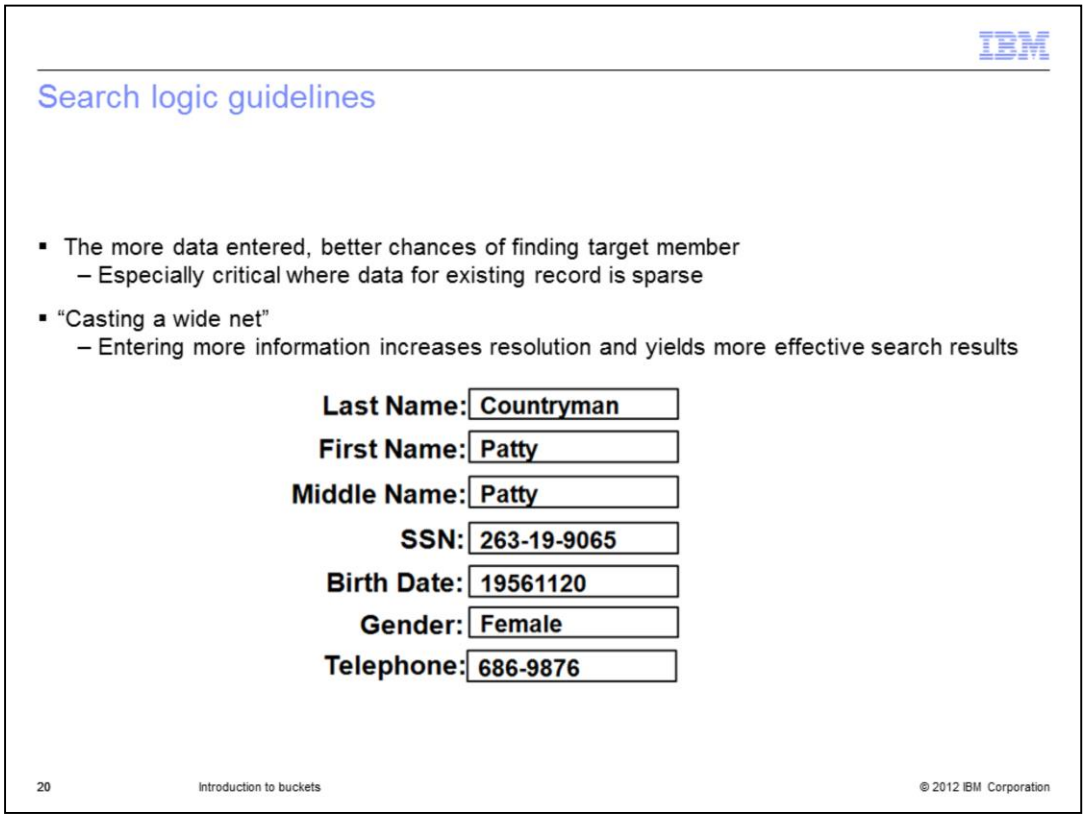



Now you are going to add the social security number to your search criteria. This time, four buckets are retrieved from the system, the previous three plus one for the social security number. When sorted, the social security number works very well by itself as a bucket.

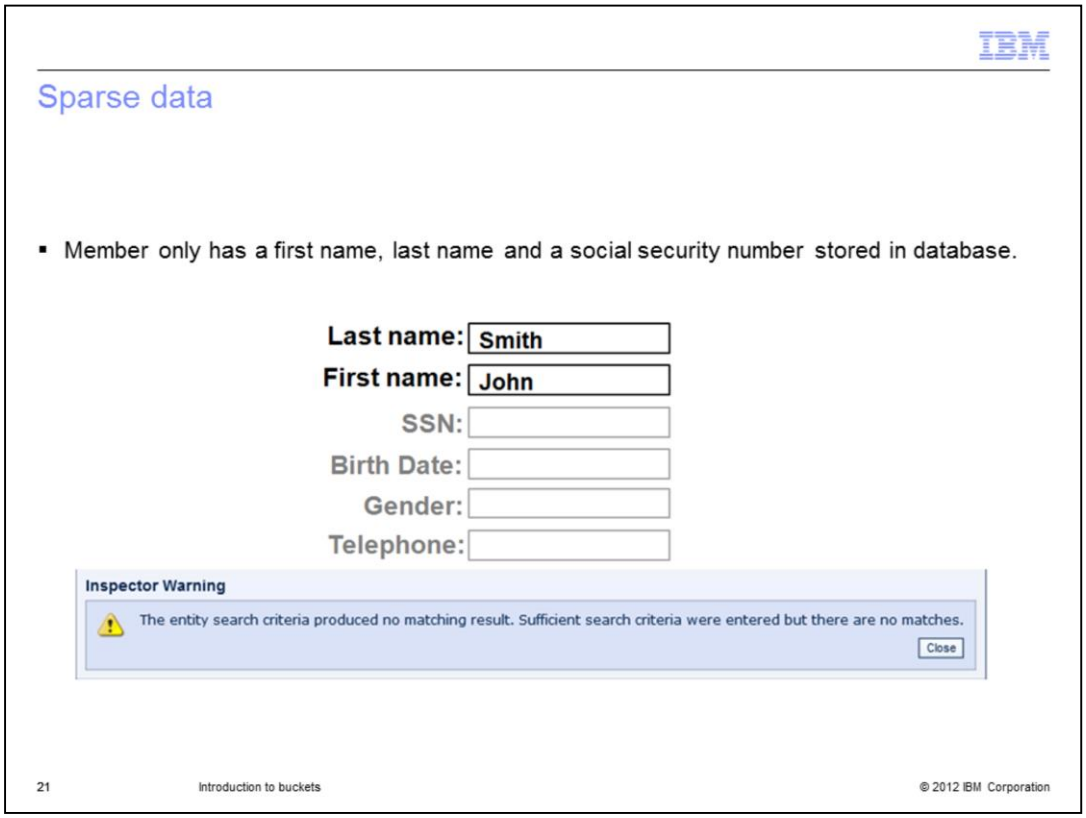
This time the total members retrieved is 4,404. There were only four members that exist in the social security bucket. The total number of members that score above one is even less than before. This time there are only 20. By adding the social security number, the number of returned members has been cut by more than half. A negative score was assigned to the social security number for the 50 previously compared members. This dropped their overall score to below 1.0.

Your member is now shown in first place in the result set. This record was the only one that contained the exact social security number match giving him the top score of 12.4.

Finally, if you enter the remainder of the information like gender, telephone number, and other attributes, you will eliminate additional members from this list. The more information you provide, the easier it is to find your intended member the first time around.




Search guidelines are very basic. You should enter as much information as you possibly can. This will allow you to retrieve as many members as possible to participate in the comparison process. You will sometimes hear reference to this as “casting a wide net”. By providing more information, you increase the chances of retrieving your target member. This is critical in cases where the search data is sparse, anonymous or blocked by Frequency Based Bucketing. Frequency Based Bucketing is discussed later in this presentation.

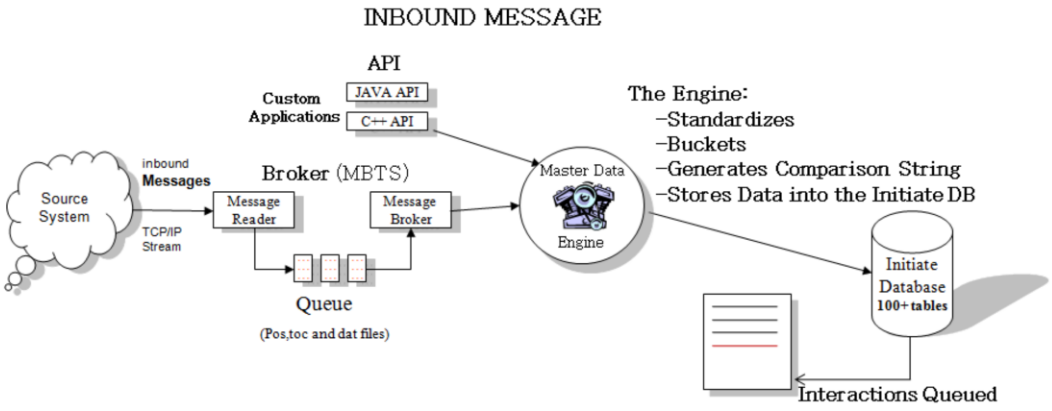



Now take a look at an example with sparse data. In this example, there is a member that only has the name and social security number attributes populated in the database. In addition, his name, “John Smith” has been configured as anonymous value due to the large amounts of members with that name in the system. No buckets exist for the “John Smith” tokens. Therefore, when you try to retrieve the member by entering his name only, no records are found, even though a member exists in the database for John Smith.

This member’s record will return in search results if the social security number is used as the search criteria. Alternately, the record can be explicitly retrieved by entering the Enterprise ID or Member Record Number in the “Retrieve Person by ID” search.

## MEMPUT logic (1 of 6)

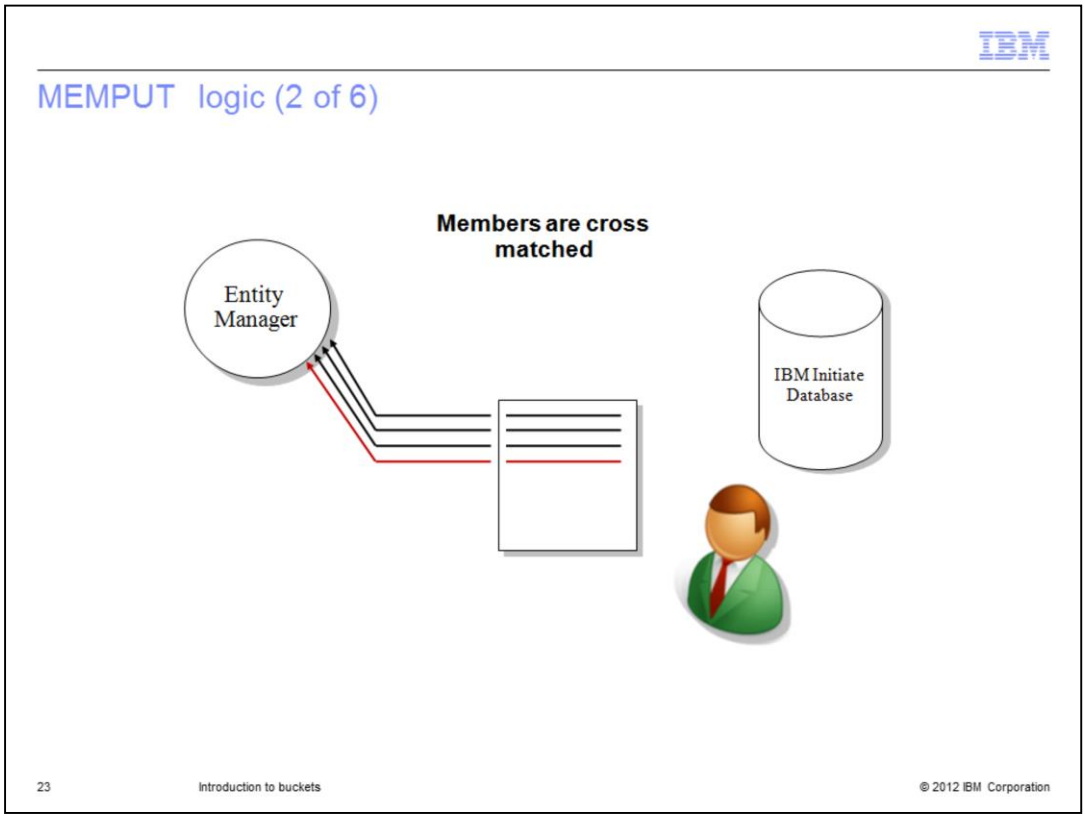
- MEMPUTs work same way as search logic
- Messages, by way of Inbound Broker or an API call
  - Similar to search criteria in search



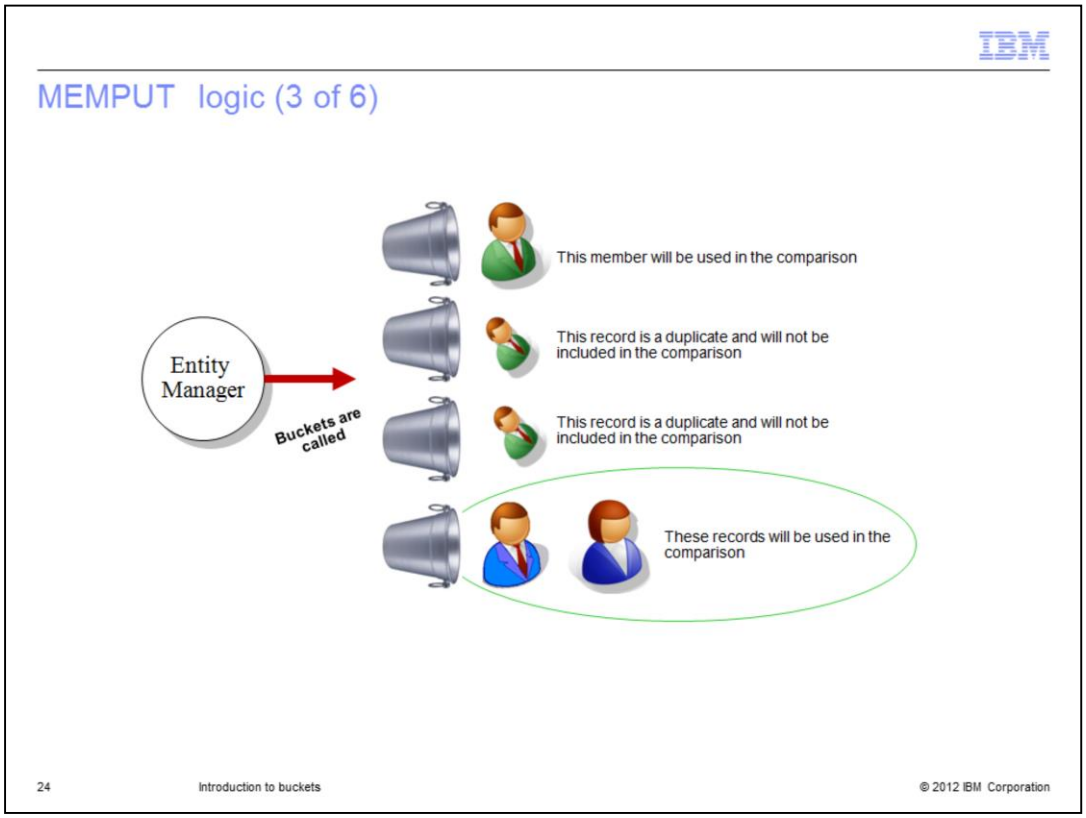




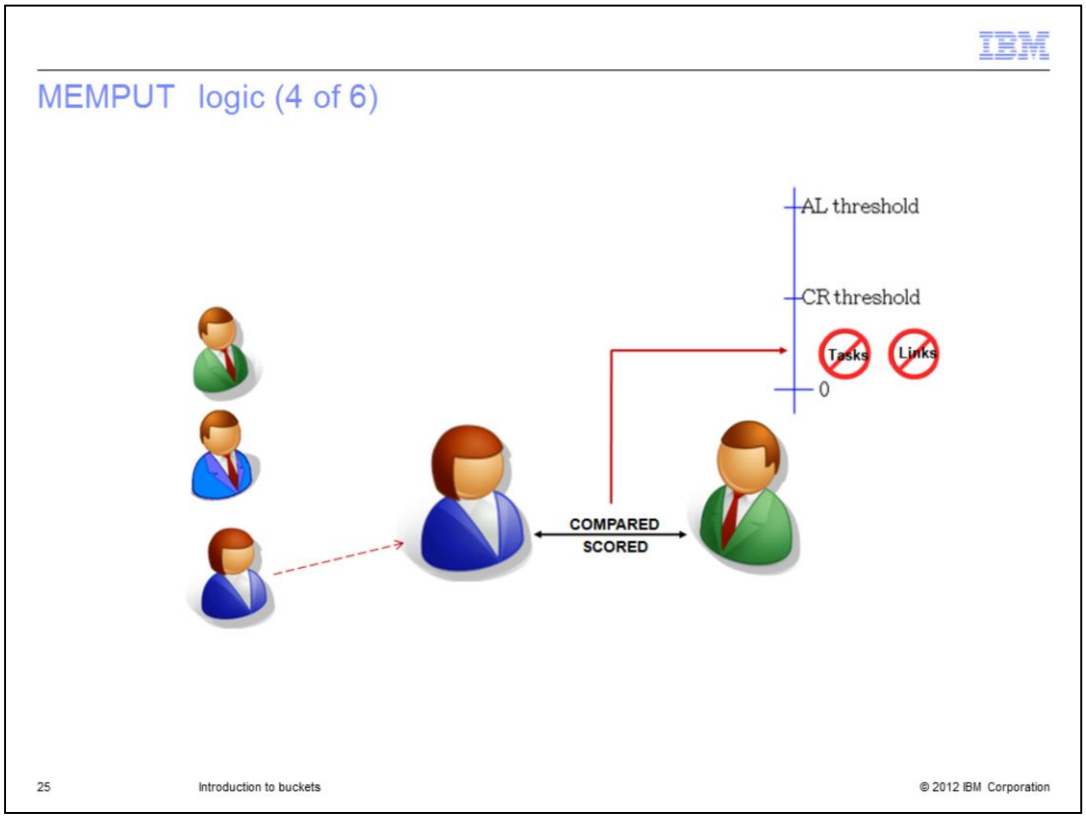
As far as buckets are concerned, MEMPUT logic works much like the search logic. However, instead of defining the search data, the custom application and broker clients provide the data. By sending in a message, the engine standardizes, buckets, generates the comparison string, and stores the data into the IBM Initiate database. The interactions are now queued to be processed by the Entity Manager.



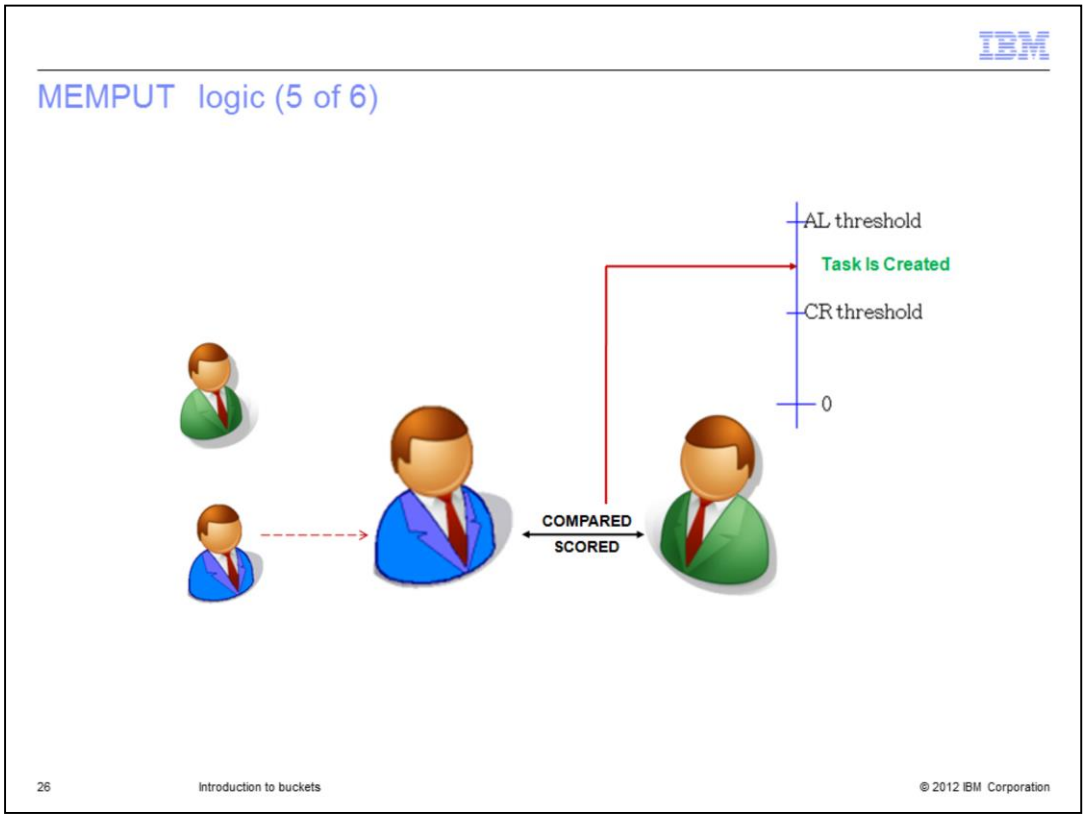

The purpose of the Entity Manager is to cross match all members in the queue in a first-in, first-out manner. For example, take the last member represented by a red arrow on this slide. You see this member's record "life cycle" from MEMPUT to successful entity management.



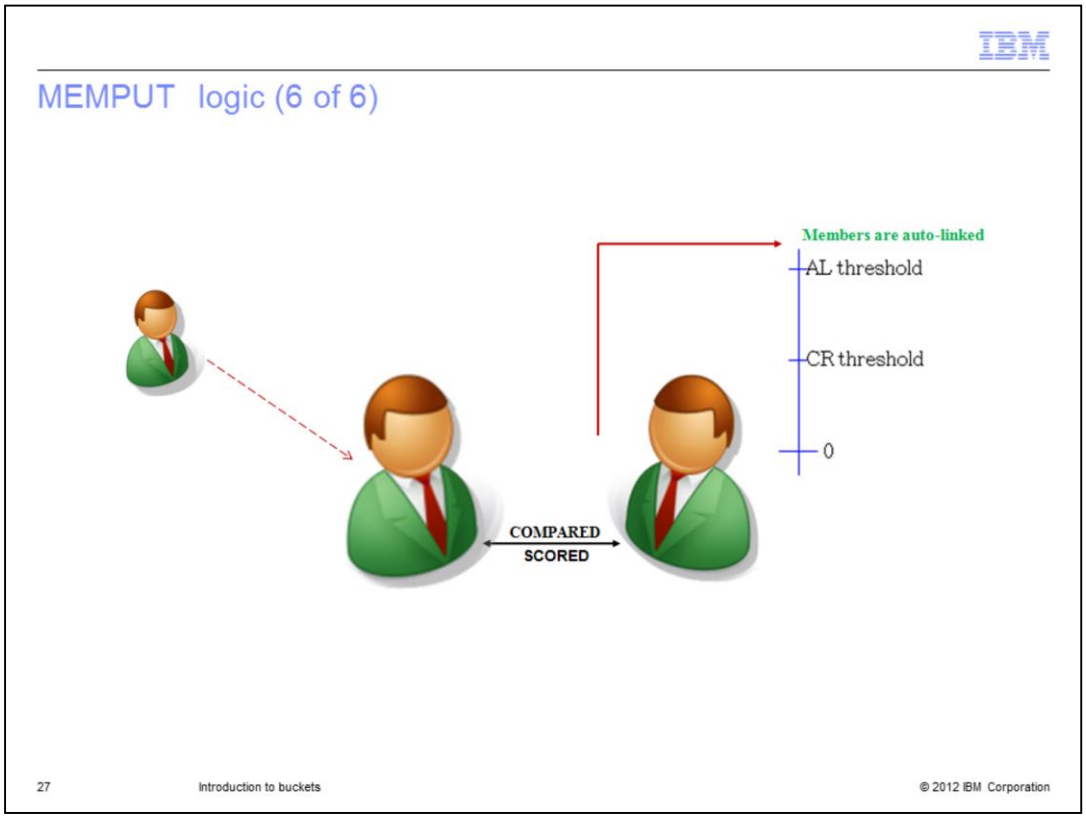

After a MEMPUT, the Entity Manager works on this member record, and the buckets associated with the record are called. The engine retrieves all members in all buckets that the target record is also in. These are called “candidate” records. Then, it removes any duplicates and uses the remaining candidates for the final similarity comparison.




The first member in the list is compared and scored. This member's score is positive but falls below the Clerical Review threshold. Therefore, no task or link is created. This member is not determined to be an applicable match to the original member.




Now the process continues to the next candidate record. It is compared and evaluated for a final score. This time the score is above the Clerical Review threshold, but below the “Autolink” threshold. Therefore, a task is created. A data steward now needs to determine if these two members are the same.



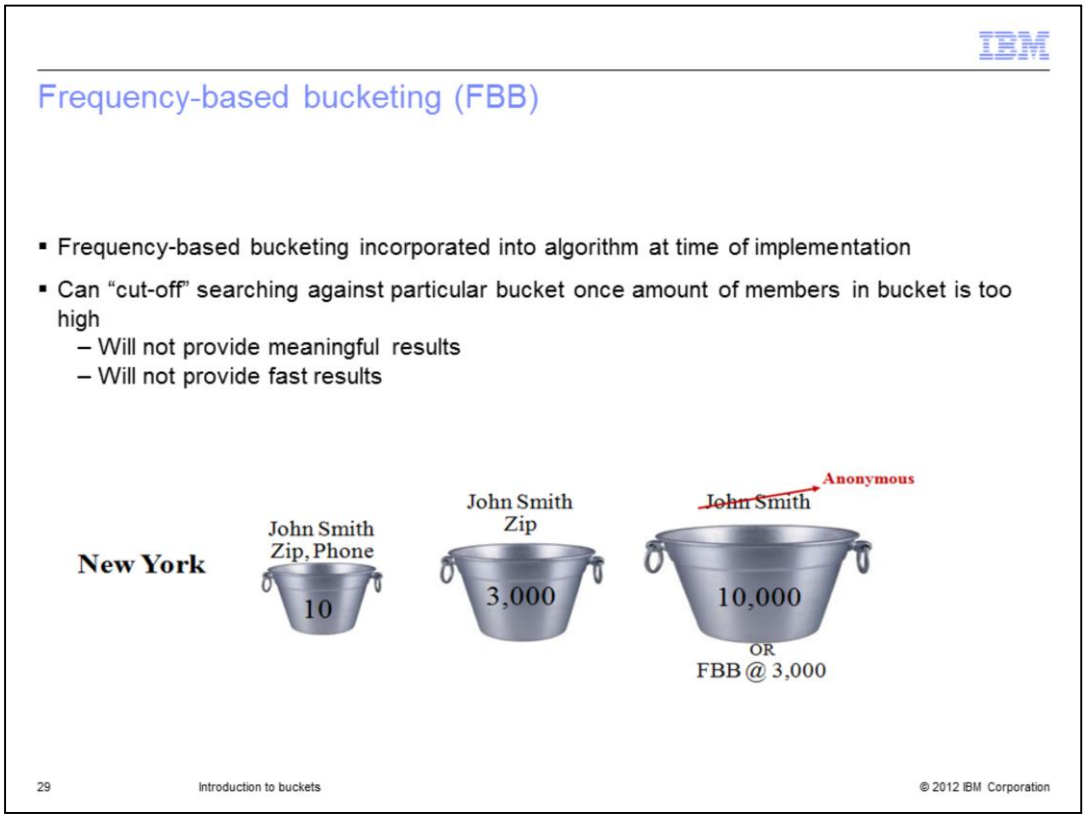

Finally, the last member in this bucket is compared and scored. The score is above the “Autolink” threshold so they are linked as the same individual. The auto linking of members saves a lot of time and manual work that would otherwise have to be performed.

## Large buckets

- Buckets grow in size as members added
- More records associated with existing buckets
- Adding of sources drastically increases buckets size
- Extremely "large" buckets can cause performance issues
  - Long search times
- Large results can render result set useless



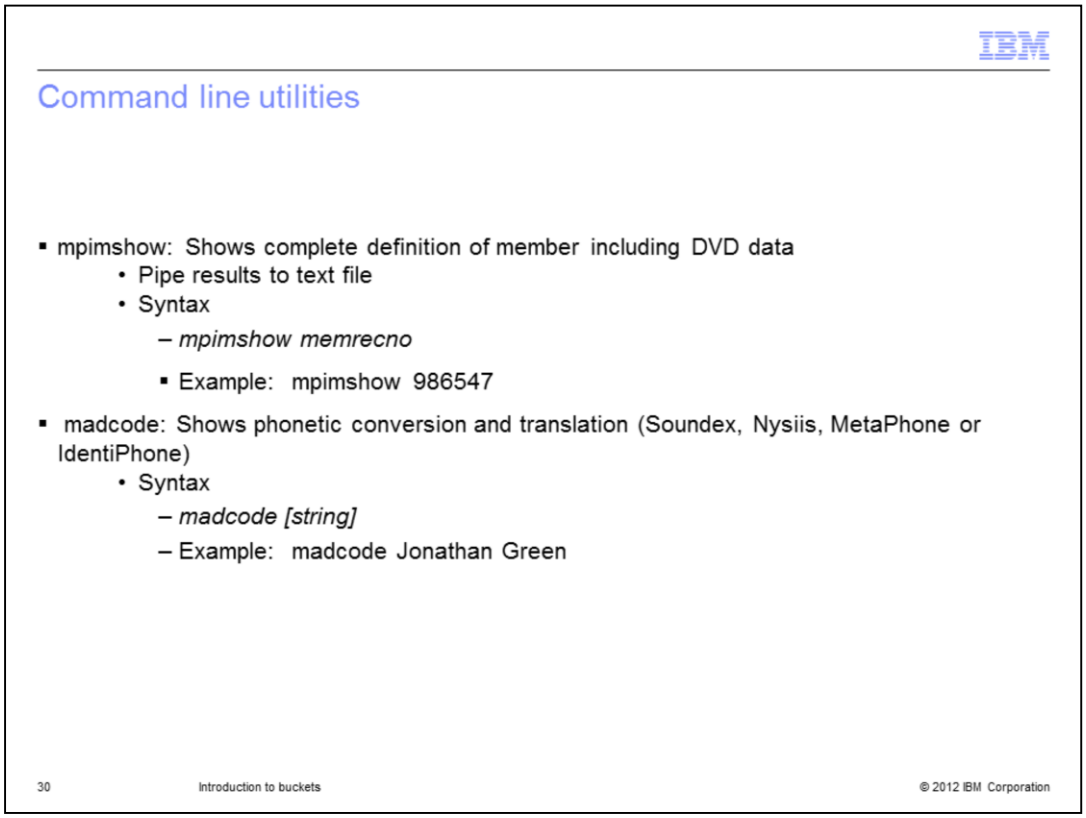

As your member base grows, by the normal addition of new members or by a bulk addition of one or more source systems, so do your buckets. Both the number of buckets in total and the amount of records associated with each individual bucket. Large buckets – those associated with thousands of member records or more - can have a negative effect on performance, and they can render search results far less useful. Earlier you saw that an example member was returned in 67th place, following a search, due to insufficient information. However, similarly poor results can occur when the buckets become too large.




One of the settings that can be configured to help alleviate the problem of large buckets is Frequency Based Bucketing. Frequency Based Bucketing, or FBB for short, is the setting of predefined thresholds that are used by the engine before performing any comparisons. This limits the growth of the bucket predefined size. At the time of your implementation and based on your demographics, the IBM services team may have performed a bucket analysis and configured FBB. They may have also made certain names anonymous where member data was deemed to be large or possessed the probability to grow at an extremely fast rate. By making certain names anonymous or limiting bucket growth to a predetermined size, you can ensure that the results of a search are fast and meaningful.

For example, imagine that you live in New York and you want to search for "John Smith". The probability of having 10,000 people that live in New York with that name is extremely high. This will give you a bucket with 10,000 members. Now you add a zip code. The number of "John Smiths" with the same zip code is approximately 3,000. This is still high. To narrow the search, add the telephone number to the existing information. Now you have "John Smith" plus his zip code plus his telephone number. The probability that a member meets all of this criteria is narrowed down to 10 members. The fact that there are 10 results can be due to typos, but as you see, the number has been greatly reduced and is now manageable.

In this case it is a good idea to either set the FBB threshold to 3,000 for this particular bucket, or, alternatively, to make John Smith an anonymous value. By configuring the name John Smith as anonymous, you will not bring back the 10,000 members. Bringing back this much data will consume computing resources and slow the system down. Moreover, the results produced by such a large search, as you have seen in this example, is useless. It will take you many hours to go through that many results for the right match. The "John Smith" bucket is not needed since you will normally be submitting additional information like zip code, telephone number and so on, in your search criteria. This eliminates the need to cross match 10,000 members, increasing speed and returning meaningful results.




**mpimshow** and **madcode** are two command line utilities that you might find useful when you are trying to understand the particular members bucket behavior or setting. **mpimshow** shows the complete definition of a member including DVD data. The syntax for this utility is “mpimshow” followed by the member’s memrecno value.

**madcode** shows the phonetic conversion/translation, which includes Soundex, Nysiis, MetaPhone, or IdentiPhone. The syntax for this is “madcode” and the name you want to see converted. For instance, if you want to know what Jonathan Green will look like converted, type **madcode jonathan green** and hit enter. A list of the phonetic translations for this name produced by each algorithm is shown.

## Related resources

- IBM Initiate Master Data Service information center (version specific)
  - IBM Initiate MDS V9.7
    http://publib.boulder.ibm.com/infocenter/initiate/v9r7/index.jsp
  - IBM Initiate MDS V9.5
    http://publib.boulder.ibm.com/infocenter/initiate/v9r5/index.jsp
  - IBM Initiate MDS V7.5 through V9.2
    http://publib.boulder.ibm.com/infocenter/initiate/legacy/index.jsp




This slide provides links to official IBM Initiate documentation online, which you can use to learn more about algorithms.

# Trademarks, disclaimer, and copyright information