

# IBM Initiate

## Introduction to algorithms



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

## Training objectives

- Understand basics of algorithms
- Understand data flow through algorithms
- Understand concept of “derived data”
- Better understanding of algorithms and use

You will gain some basic information on the algorithms and what the data flow looks like as it is processed. You will also be introduced to the concept of “derived data”. After this presentation, you should have a better understanding of the algorithms and how they are used to manipulate data.

## Algorithm basics

- Dictionary.com's meaning for word "algorithm":
  - *"a set of rules for solving a problem in a finite number of steps, as for finding the greatest common divisor."*
- Initiate algorithm
  - Series of computational processes that analyze member records
  - Brain behind probabilistic matching process
- IBM Initiate Master Data Service® algorithm has three steps
  - Standardize
  - Bucket
  - Compare

### What is an algorithm?

An online dictionary's definition for "algorithm" is given as: "a set of rules for solving a problem in a finite number of steps, such as for finding the greatest common divisor". This is a good high level explanation of how the algorithm is used in the IBM Initiate software.

An IBM Initiate algorithm, is a series of computational processes that analyze member records. For the IBM Initiate Master Data Service, the algorithm is the true key to your probabilistic matching process.

IBM Initiate software applies proprietary algorithms to compare and score member attribute similarities and differences. The IBM Initiate Master Data Service algorithm has three steps: standardize, bucket, and compare. The final step, compare, is what yields the score. The algorithms are applied to data to create tasks and support search functionality, and are tailored to use attributes specific to your business.

## Why are algorithms needed?

- Accessing data
  - Searches
  - Additions
  - Updating data
- Business needs
  - Algorithm is unique
  - Different demographics

### Why are algorithms needed?

The algorithm is used to optimize searches, additions, and updates to your existing data. It allows your system to be configured in order to meet your business needs based on your data. Everything in the algorithm is based on your data, so each algorithm is unique for every customer.

Depending on the demographics, customers often have radically different setups from each other. Therefore, an algorithm configuration cannot be copied from one dataset to a different one.

## Glossary terms

- Anonymous values - To accommodate occurrence of missing member data without adversely affecting comparison score
- Candidate - List of member records that meet search parameters
- Comparison score - Score returned for records after comparison process performed
- Core member data - Complete set of data (all attributes) for a member

This slide displays terms that you need to know as they are mentioned through-out this presentation.

**Anonymous values.** Anonymous values are a set of values that are added to the system. These values are specifically not included in the comparison. For example, if a record is being entered into a source system and the social security information is not available, the registration clerk can enter a value of all zeros or all nines. In such a case, if these numbers are seen in the social security number field, they are not considered by the algorithm in the comparison because they are bogus. The algorithm is configured to not assign a score to attributes containing anonymous or missing values.

The next term is candidate. A candidate is a single result of a search request. When a search request is sent to the Master Data Engine, the engine returns a list of member records that meet the search parameters. These records are referred to as candidates.

Comparison score is the score that is returned for each candidate. Every candidate that returns from the candidate selection process will have a score associated to it. It represents how close it matched your search criteria. The higher the score, the more likely that the compared records represent the same person. Similarly, when you add a member, every record that matches this new member will have a comparison score associated to it.

Core member data is the complete set of data, which includes all attributes, for a member. It is stored in the Initiate database without modification. All data received from the system is stored in the database exactly as it was received. The process of how to break this information down is talked about later.

## Algorithm - overview

- Standardization
  - Converts data to consistent format
  - Ignores anonymous values
  - Keeps copy of standardized comparison data
- Bucketing
  - Groups records that share common attributes
  - Used for candidate selection (possible matches)
- Comparison
  - Standardized attributes are compared and weights assigned to each attribute's result
  - Score is generated and used as result of comparison

This slide displays an overview of algorithms. At a high level, algorithms consist of a standardization, bucketing and comparison process. A detailed look into each process is covered in the next slides.

The first process that takes place is standardization. When data is standardized, it is made unique and easier to manage. During the standardization stage, the data is modified to create a consistent view. For example, removing apostrophes and capitalizing characters.

In addition, anonymous values are ignored, such as in the second example where the social security number consists of all nines.

In the last step of the standardization process, the standardized data is stored in the database. It is worth noting that the software will store both a copy of the original data as it was received and the new standardized data. The standardized data is used during the matching and linking process, where the original data is used for display to the user.

The next step is bucketing. A bucket is a group of records that share common attributes. It is used for candidate selection to find potential matches. As an example, imagine a play room that has toys organized into various buckets. In one bucket you have building blocks and in another you have cars. If you are looking for a specific toy car to play with, you know immediately which bucket to look in. Compare this to one giant bucket that contains all the toys and you can see how this form of organization can greatly increase performance.

Finally, there is the comparison process. In the comparison phase, the standardized attributes are compared and weights are assigned to each attribute's result. Once weights are calculated, the comparison score is generated and returned as the result. Going back to the toy car example, this is the part of the process where you go through the bucket of cars and rank each one based on what attributes you are looking for. For example, a blue race car with large wheels.

In summary, for the algorithm process, first the data is standardized, then buckets are created and finally the data is compared to get the final scores.

## Standardization basics

- Used to convert data into more usable format for searching and comparing
  - Convert data to standard form
    - Converting to upper case
    - Removing extra characters (spaces or dashes '-' in telephone numbers)
  - Filter out “bad” values (phone number 00000000)
    - Length
    - Anonymous values
  - In some cases, generate a list of tokens
    - Personal name “JOHN B ADAMS -> ADAMS:JOHN:B

Standardization is the process where by data received, in various formats, is transformed to a more uniform format that enhances the comparison process. The incoming data is stored in both the newly standardized format and in its original format.

The standardization process converts all characters to be upper case and removes extraneous special characters, such as dashes or spaces. It filters out predefined bad values, like the anonymous values mentioned in a previous slide. In this example, the bad value is a telephone number of all zeros.

The process can also filter out any words that do not meet a specified character length. A good example of this is a North American social security number which is defined as being nine digits long. Anything other than nine digits is not considered a valid social security number.

In some cases, you will break apart an attribute into what are called ‘tokens’. Tokens are commonly used with names and addresses. For example, the personal name JOHN B ADAMS is split into three tokens, ADAMS, JOHN and B. This will allow the creation of separate buckets and perform comparisons on each token individually.

## Standardization functions (1 of 2)

- AGE – Removes non-integers, ignores invalid ages
- ATTR – Removes non-alphanumeric characters
- BXNM – Standardize business names
- CNADDR, USADDR – Canadian or US addresses
- CNZIP, USZIP – Canadian or US postal codes
- DATE(1,2), GRDATE – Standardize dates
- EMAIL – Checks for “@” and “.”, removes everything after “.”

This slide displays a list of some common standardization functions. Within each function, there are several configuration options to allow each customer customization in order to meet your needs. As you can see, it covers a wide range of data types.

The “AGE” function removes non-integers and ignores invalid ages.

The “ATTR” function removes non-alphanumeric characters.

The “BXNM” standardizes business names.

The “CNADDR” and “USADDR” functions are for Canadian or United States addresses.

The “ZIP”, “DATE” and “EMAIL” functions are self explanatory.



## Standardization functions (2 of 2)

- EYECOLOR, HAIRCOLOR, RACE – Same as ATTR
- HEIGHT – Converts height in FII format to inches
- IDENT(1, 1N, 1A) – Same as ATTR but also checks for valid length, can pre-append source code to ID
- PHONE(1,2,END) – Standardizes telephone numbers
- PXNM – Separates names into tokens, combines tokens when desirable
- WEIGHT – Removes non-integers, ignores invalid weights
- ATTRN - Removes non-alpha
- ATTRA - Removes non-numeric

The “EYECOLOR”, “HAIRCOLOR” and “RACE” functions are also self explanatory.

The “HEIGHT” converts a height in FII format to inches.

The “IDENT” is the same as ATTR but also checks for valid length and can pre-append source code to ID.

The “PHONE” function standardizes telephone numbers.

The “PXNM” function is commonly used on personal name.

The “WEIGHT” function will remove non-integer values and will ignore invalid weights.

The “ATTRN”, and “ATTRA” functions are used to remove non-numeric or non-alphabetic characters.

## Standardization example 1 - Social security number

- social security number is standardized using IDENT1N
  - Removes all non-numeric
    - 294-44-1234 becomes 294441234
  - Filters out all social security numbers whose length is not 9
  - Filters out all anonymous social security numbers
    - 000000000, 111111111, 123456789

The examples in the next slides details the utilization of standardization functions and walks you through each step of the process.

The first example will employ the IDENT1N standardization function. The IDENT1N function removes all characters which are not numeric. In this example, the IDENT1N standardization function removes all the dashes and any spaces. In addition, it filters out any social security numbers whose length is not nine digits long and any anonymous values. A couple of anonymous examples are a social security number containing all zeros or a consecutive number such as 123456789.

## Standardization example 2 - Date

- Dates can be standardized using DATE1 function
  - DATE1 removes non-numeric characters
  - Checks length of value
  - Checks remaining string for valid date
    - 19710231 not valid
  - Removes anonymous values
    - 19000101

In this example, a date is standardized. Date values can be standardized using the DATE1 function.

The DATE1 function removes any non-numeric characters and then checks the length of the date value. Any value that has a length of zero is treated as an anonymous value. Any value that has a length greater than eight digits is truncated to include the first eight characters only.

The DATE1 function then checks that the remaining string is a valid date. In this example, 19710231 is not valid since February does not have 31 days.

As with the previous social security number example, it also removes any anonymous values. The anonymous value list depends on demographics. For example, you may have the need to include 19000101 as an anonymous value, and if you did use that as an anonymous value, this date is filtered out and ignored.

## Standardization example 3 – Personal name

- Personal name standardized using PXNM
  - Creates list of tokens
  - Appends common prefixes such as MC (MC LANE -> MCLANE)
  - Filters out anonymous tokens such as BABY or UNKNOWN
  - Filters out titles, degrees, and so on (MR, MRS, MBA)
  - Separates suffixes: JR, II, III
    - Used in comparison
      - Mr Bob Q D'Angelo II becomes DANGELO:BOB:Q:.II

The last example for standardization uses personal name. Typically, personal names are standardized using the PXNM function.

The PXNM function creates a list of tokens, appends common prefixes, filters out any anonymous values, titles and degrees and separate suffixes. Suffixes are typically used in comparison since these help differentiate people in the same family. For example, identifying a father from a son who both share the same name.

In this example, Mr Bob Q D'Angelo II, will become DANGELO:BOB:Q:.II after standardization.

## Bucketing

- Part of derivation process in which Master Data Engine groups specify attributes for use by candidate selection process
- Record only compared against records that it shares buckets with
  - Most records will have multiple buckets
- Examples of possible buckets
  - Last name + phone
  - Last name + email address
  - Last name + first name
- Buckets defined during initial configuration of Initiate Master Data Engine by Initiate project team
  - Can have one or more attributes per bucket

The second process in an algorithm, is the bucketing process. Bucketing is a part of the derivation process in which the IBM Initiate Master Data Engine groups members together based on matching specific attributes.

An important note is that a record is only compared against other records that it shares a common bucket with. Most records will have multiple buckets.

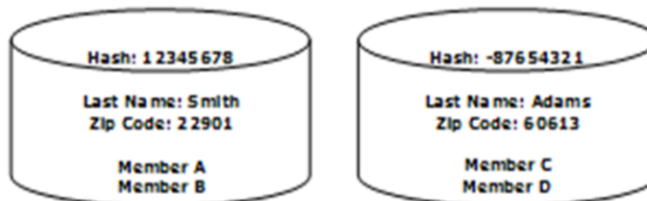
Previously in this presentation, the example of the play room was shown. The example consists of several buckets in which to put sets of toys in. There is one bucket with building blocks, another bucket with cars and another one with dolls. If you are looking for a specific toy car, you will not look in the bucket that contains dolls. Instead, go directly to the bucket that contains cars. In this example, the attributes that got bucketed were the type of toys.

Some other examples of possible buckets is last name and a telephone number or, last name plus the email address or, last name plus the first name.

The idea is to generate buckets for data that can be shared by several records. When you do a search and find that several records share buckets, you can tell that those records share some similar attribute data and are worth adding to the candidate selection process for comparison.

## Bucketing basics (1 of 2)

- Records that share common attributes are grouped together into same bucket using hash value
  - For example, all records who share last name and zip code are placed into same bucket
- Initiate engine uses buckets to find candidate records similar to search criteria
- Uses comparison functions to determine which candidates are matches
- Sometimes referred to as “casting a wide net” for possible matches



14

Introduction to algorithms

© 2012 IBM Corporation

To drive the point home it is important to look a little closer.

As stated earlier, records that share common attributes are placed together in the same bucket using what is known as a ‘hash value’. For example, all records that share a last name and zip code might be placed in the same bucket.

Consider this example. A hash value exists for the last name of “Smith” with a zip code of “22901”. There are two members who have this data in common, Member A and Member B.

In another bucket, there is a different hash value. This one represents a last name of “Adams” with the zip code of “60613”. This bucket houses Member C and Member D.

The engine uses buckets to find candidate records that are similar to your search criteria. When you go into your system and look for “Smith” and zip code “22901”, the IBM Initiate system then converts the search into a hash value which is going to match the “12345678” hash value. Instead of doing the string search, a number or “integer” search is performed, since this sort of search is a lot faster for databases. The search will retrieve Member A and Member B, which share the bucketing information. The same thing is done when searching for “Adams” with the zip code “60613”.

The results that are returned are called candidates. The idea is to “cast a wide net” for potential matches based on one or more similar attributes. Effectively, narrowing the search down and increasing the speed, instead of having to compare against all the members in the database. Once the candidate selection list has been obtained, the comparison is performed on each one to get the scores.

## Bucketing basics (2 of 2)

- Bucket values derived from standardized data and converted into hash value
- Standardized data transformed based on number of bucketing functions
- Standardized values can be combined into single buckets

The bucket values themselves are based on standardized data which is converted to and stored as numerical hash values. The standardized data is transformed using bucketing functions. Multiple standardized values can be combined into a single bucket.

## Example bucketing functions

- ASIS – Use values unmodified
- META1,META2,META3,META4 – Use phonetic conversion (metaphone, prefixmap, arabphone, or identaphone)
- EQUI – Use nickname translation
- EQMETA – Same as EQUI + META

This slide displays some of the available bucketing functions. For “ASIS”, the standardized attribute is used without additional modification.

With “META”, a value can be translated to its phonetic equivalent, a few examples of “META” functions are covered later.

Next is “EQUI”. This one is used for nickname translation. For example, the name Robert can have multiple nickname possibilities such as Bobby, Bob or Rob. So if someone is looking for Bob Smith, but in the database you have Robert Smith, the record can still be found.

“EQMETA” is a combination of phonetic conversion using a nickname. This function converts the nickname to it’s phonetic version. There are many more bucketing functions available for use.



## Designing a bucket strategy

- Bucket design is an art
  - Creating buckets that yield very specific data (small buckets) will not allow efficient search of records
  - Creating buckets that are too generic will return less useful results and eventually affect system's performance
- Buckets must support searches
  - If searching on name alone, you must have name-only bucket configuration

Now that you are familiar with how buckets are used, you now need to learn how they are created.

Designing a bucket strategy is an art. It is all about looking at your data and understanding how to access it most efficiently and for the greatest results.

Creating a large number of small buckets based on specific data will not allow an efficient search of records. Going back to the play room example, this is the same as breaking your car bucket into several small buckets such as a bucket for red trucks made in 2005 with four doors. This makes it so that if you are wanting to search for any red truck, you will not be able to find it since there is no single bucket that contains red trucks.

The opposite extreme is creating buckets that are too generic. These become large buckets that return less useful results and slow down performance. This is the same as having a single bucket for all cars. If you are looking for a red truck, it will take some time to go through the entire bucket to see which ones match your criteria.

This is how you want to design your buckets if you know that you plan to search for your toy cars based on the type of car and the color.

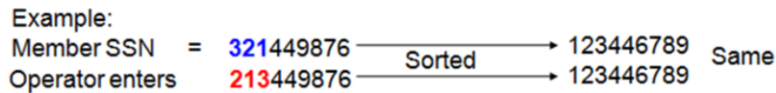
The idea is that you want to make a search as efficient as possible while still getting the desired results. One method to ensure effective results, is to make the data more generic by way of methods like phonetic and sorted. However, the data cannot be too generic as to create huge buckets, such as a one name token. Common names like "John" or "Mary", create huge buckets. Depending on locale and demographics, those buckets can be a few thousand or hundreds of thousands of records in size. That does not help narrow down the results at all and will greatly consume computing resources. Doing multiple name token buckets to retrieve "John Williams", for example, will help narrow the results and increase performance while still offering a fairly wide net of candidate records to compare.

An important note, not always obvious, is that buckets must support searches. Any type of search that you perform needs the data to be bucketed. If you want to search on two name tokens only, you need to have a two token name bucket setup in your algorithm.

## Bucketing example 1 – social security number

- When sorted, social security number by itself works well as a bucket
- Bucket generation type of “SORTED”

Example:  
Member SSN = 321449876 → 123446789  
Operator enters 213449876 → 123446789 Same



This slide displays an example for the social security number bucketing function.

“SORTED” is used as the bucketing function since this helps broaden the candidate selection.

In this example, you can see that the social security number is transformed to be in numerical sequence order “123446789”. This resolves the problem of users accidentally transposing two or more numbers. For example, a member’s social security number might be “321449876”. If the operator inputs the digits in the wrong order by mistake, say “213449876”, that is okay because once they are sorted, they are the same.

You may have noticed that there is a good chance you can match on other records with the same sorted number. Do not worry about that quite yet, this is only used for bucketing to get the candidate selection. There is still the comparison step which will order the results based on a score.

## Bucketing example 2 – DOB + Phonetic name

- Date of birth alone can potentially create large buckets because of high number of DOB records
- Typically, DOB is combined with name or phonetic name token
  - Example: **JOHN SMITH** with DOB **1971-03-24**
    - Phonetic codes are JN and SMZ
    - Generated bucket values
      - 19710324+JN
      - 19710324+SMZ

Another example is date of birth plus a phonetic name.

Date of birth by itself can potentially create large buckets due to the high number of records sharing the same birth year. Typically, the date of birth is combined with a name or phonetic name.

This example depicts a name “JOHN SMITH” combined with date of birth “1971-03-24”. Since you are doing the phonetic version of the name, “JN” is substituted for “John” and “SMZ” for “Smith”. Based on this, the bucket permutations are the date of birth plus “JN”, and the date of birth plus “SMZ”.

## Bucketing example 3 – Name pairs with nicknames

- Another common bucket – Name Token Pairs using nicknames
- Example: BILL STEVEN SMITH will create buckets
  - BILL+STEVEN
  - STEVEN+WILLIAM
  - BILL+SMITH
  - SMITH+WILLIAM
  - SMITH+STEVEN
- Bucket values always alphabetical

This slide displays another example for name pairs with nicknames.

If you recall, back on the standardization function on names, the PXNM function was used to break up the names to one or more tokens.

This example uses the name “BILL STEVEN SMITH”. Two token permutations results in these buckets:

BILL plus STEVEN

STEVEN plus WILLIAM – remember, WILLIAM now is a nickname

BILL plus SMITH

SMITH plus WILLIAM

SMITH plus STEVEN

It is worth noting that the tokens are always in alphabetical order so you do not have repeated buckets.

## Comparison

- Similarities and differences between two or more records
- Compares individual attributes—attribute by attribute—of each record selected during candidate selection process

Comparison is the step of the IBM Initiate algorithm where similarities and differences are determined between two records. Once the candidate selection list has been obtained from the bucket results, the comparison process is initiated. It compares the data used for the search criteria or triggers member data for each candidate. This is done by comparing each of the attributes as they were standardized and then provides the comparison score.

## Comparison basics

- Compare Functions compare standardized data and assigns “score” to result
- Positive score means result is evidence for a match

As each attribute is compared, a score is assigned based on the weight values defined in the configuration. Once all attributes for a given comparison have been processed, the scores are tallied and a final score is produced.

The results are then returned to the calling client application with the highest score listed first. The results are typically displayed in descending order, based on score. The idea is that the top result, the result with the highest match score, is the most likely candidate.

## Comparison functions

- DR - Edit distance functions
- EQVD - String equivalence
- EQVN - Integer equivalence
- USZIP - Zip code comparison
- DATE - Date comparison
- PXNM, QXNM - Person name comparison
- CXNM - Business name comparison
- BXNM - Provider name comparison

This slide details a list of some of the comparison functions.

The first one is actually a group of edit distance functions. With these, you can compare one string to another in order to find out how many changes are required for one string to be the same as the other string.

The next two, string and integer equivalence, state whether the string or integer, respectively, are an exact match or not.

Zip code and date comparison functions each have their own rules. The “DATE” function, for example, will return weights based on a year only match, versus year, month and day match. In addition, there is person name, business name and provider name comparison and many more.

## Comparison example 1 - social security number

- social security numbers compared using Edit Distance function which measures number of character differences
  - Examples
    - 123456789 – 123456789 are identical, have edit distance of 0
    - 123456789 – 223456789 are off by single character and have edit distance 1
    - 123456789 – 213456789 are off by single transposition and have edit distance 1

The first comparison example is a social security number example. Social security numbers are typically compared using an edit distance function. You can say that this describes the “amount of difference” between one value and another, or the number of actions for one value to be made identical to the other.

Therefore, in the first comparison, the two records are identical which means they have an edit distance of zero. In the second comparison, the numbers are off by a single digit, therefore, they have an edit distance of one. The first digit “1” needs to be replaced by a “2” to make them an exact match.

In the last comparison, the two strings are off by a single transposition, most likely caused by someone transposing the two numbers. This too will have an edit distance of one as in the previous comparison. The reason, is that it only takes one swapping of the digits to make an exact match.



## Comparison example 2 - Date

- Dates compared using edit distance
- If exact match exactly, weight is determined by birth year
  - Rare birth years have higher weight than common birth years
- Example
  - In most populations, birth year 1910 is rarer than 1971
  - Exact match for birth date in 1910 will score higher than 1971

Dates are also compared using edit distance, as this presentation illustrates. If they match exactly, the weight is determined by the birth year. In this example, there are two years, 1910 and 1971. In most customer databases, it is less common to see someone who was born in 1910 than in 1971. Therefore, a match on the year 1910 results in a larger weight being used than a match on 1971.

It is worth noting that while an initial baseline set of weights is provided, once customer data is loaded, customers typically go through a weight generation process to customize the weights to match the customer's dataset. For example, if one customer deals mostly with infants, and another deals mostly with the elderly, the majority of the birth years will look significantly different, and so should the corresponding weights.

## Comparison example 3 – Personal name

- Personal names compared token by token for all possible combinations
- First test is for an exact match
- Failing that, the tokens are checked for
  - Phonetic matches (JOHN-JON)
  - Nickname matches (BILL-WILLIAM)
  - Initial matches (G – GREG)
  - Edit distance matches (FREDRICK-GREDRICK)
- Example
  - BILL G JONES – JIMMY WILLIAM JONES
  - BILL-WILLIAM – Nickname (+1.13)
  - JONES-JONES – Exact (+2.13)
  - G – JIMMY - Disagreement (-1.50)
  - Total score = +1.76

A personal name example is also described. Personal names are compared using all permutations of available tokens. The first test is for an exact match. If that fails, the tokens are checked for phonetic matches, such as “JOHN”, J O H N, and “JON”, J O N nickname matches. Or such as “BILL” and “WILLIAM”, initial matches. Also, as in ‘G’ for “GREG”, and edit distance matches, such as “FREDRICK” and “GREDRICK”.

For example, “Bill G Jones” versus “Jimmy William Jones”. “Bill” and “William” will have a nickname match, which will produce a positive score. “Jones” and “Jones” goes for an exact match, which will produce an even higher positive score. However ‘G’ and “Jimmy” are at a disagreement, there is not even an initial match. This one produces a negative score, therefore, for this example, your total score for the personal name comparison is 1.76.

## Data flow – Adding records

- Add new record
  - Record for “Kevin O’Malley” is added
- Standardize data
  - Kevin O’Malley changed to KEVIN OMALLEY
- Generation of Buckets and Comparison String
  - Hash values generated and stored
  - Comparison string generated and stored
- Search and compare
  - Searches for similar records using buckets
  - Compares new record to all matched records
  - Creates linkages and tasks based on comparison score results

Now, the focus is diverted towards data flow. What does the data flow look like when you add a record? The first thing that happens is you add a record for “Kevin O’Malley”. Then the record is standardized, therefore, “Kevin O’Malley” is changed to “KEVIN OMALLEY”, using all capital letters and leaving out the apostrophe after O. Next, buckets and comparison strings are generated and stored. Finally, the record is compared against all the candidates returned. Based on the resulting scores and your configuration, linkages and tasks are created and stored.

## Data flow – Searching records

- Search for record
  - Search for Kevin O'Malley performed
- Standardize search data
  - Kevin O'Malley changed to KEVIN OMALLEY
- Generation of Buckets and Comparison String
  - Hash values generated
  - Comparison string generated
- Search and compare
  - Searches database for generated hashes
  - Compares search record to all returned records
  - Returns results based on comparison score

Here is what the process looks like for a search. Say you are in IBM Initiate Inspector or some other application that is allowed to perform searches and you do a search for “Kevin O’Malley”. The first thing the algorithm does is it grabs the search data and standardizes it. Once standardized, the bucket hash values and comparison strings are generated. Now bucket hashes that match the generated ones are retrieved from the database. These returned matching hash values are effectively the candidate selection. The algorithm then compares each of the candidates against search criteria. Then the returned results are going to be ordered based on the comparison score. The one with the highest score is the first one returned, and the lowest score is pushed down the list.

## Warning

- Algorithms are at heart of your IBM Initiate Master Data Engine
- Strongly encouraged to contact IBM Support if you want to make any changes
- At very least, changes should be made and thoroughly tested in environment completely separate from production system

Algorithms are at the heart of your IBM Initiate Master Data Engine. You are strongly encouraged to contact IBM Support if you feel you want to make any changes. At the very least, changes should be made and thoroughly tested in an environment completely separate from your production system.

The reason this is cautioned, is because changes to your algorithms will affect your system. If you make a change that affects the large majority of the records to create higher score for a certain set of data, then you might generate a large number of tasks or a large number of unwanted linkages. Changes made can also negatively impact system performance. If you think a change should be made to your algorithm, contact IBM Support and you will then be given some tips on the correct steps to make changes. IBM Support can also arrange for an IBM consultant to work with you in order to perform analysis on your data and make recommendations, if necessary.

## Related resources

- IBM Initiate Master Data Service information centers (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

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](http://www.ibm.com/legal/copytrade.shtml)" 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.

© Copyright International Business Machines Corporation 2012. All rights reserved.